AD-A019 334

A KNOWLEDGEABLE, LANGUAGE-INDEPENDENT SYSTEM FOR PROGRAM CONSTRUCTION AND MODIFICATION

Martin D. Yonke

University of Southern California

Prepared for:

Defense Advanced Research Projects Agency

October 1975

ARPA ORDER NO. 2223

ISI/RR-75-42
October 1975

Martin D. Yonke

A Knowledgeable, Language-Independent System for Program Construction and Modification

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA ISI *4676 Admiralty Way/ Marina del Rey/ California 90291*
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RR-75-42	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Knowledgeable, Language-Independent System for Program Construction and Modification		5. TYPE OF REPORT & PERIOD COVERED Research Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Martin D. Yonke		8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C 0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order # 2223 Program Code 3D30 & 3P10
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE October 1975
		13. NUMBER OF PAGES 68
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) -------		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

----

18. SUPPLEMENTARY NOTES

------

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Interactive editing, language-independent, structure-oriented editing, programming environments.

✓ A

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT

The need of a language-independent programming environment with knowledgeable facilities is explicated. Then the design of a language-independent system for "intelligent" creation and modification of programs is presented as an example of such a facility. This system, called the Program Constructor and Modifier, is a two-stage process. In the first stage, an "expert" creates a description of a programming language in a high-level formalism. This description is used in conjunction with the underlying model of programming languages to drive the second stage, in which the general user creates and modifies programs written in the particular programming language. This model will guarantee that throughout the interaction the program is syntactically error-free and -- as far as possible without executing the program -- will guarantee certain semantic consistencies. All methods associated with this model are oriented towards error prevention while still allowing the user "free-form" program input. These methods will also automatically correct certain classes of errors such as misspelled words and omitted terminal symbols of certain types and will interact with the user to gain information when there is insufficient knowledge for automatic correction.

This is part of a series of reports describing ISI research directed toward reducing significantly the cost of military software while improving its application and upgrading the general quality of software. This report covers a significant portion of the author's University of Utah doctoral dissertation, completed at ISI.

ARPA ORDER NO. 2223

ISI/RR-75-42
October 1975

Martin D. Yonke

A Knowledgeable, Language-Independent System for Program Construction and Modification

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308. ARPA ORDER NO. 2223 PROGRAM CODE NO. 3D30 AND 3P10

CONTENTS

ABSTRACT

The need of a language-independent programming environment with knowledgeable facilities is explicated. Then the design of a language-independent system for "intelligent" creation and modification of programs is presented as an example of such a facility. This system, called the Program Constructor and Modifier, is a two-stage process. In the first stage, an "expert" creates a description of a programming language in a high-level formalism. This description is used in conjunction with the underlying model of programming languages to drive the second stage, in which the general user creates and modifies programs written in the particular programming language. This model will guarantee that throughout the interaction the program is syntactically error-free and -- as far as possible without executing the program -- will guarantee certain semantic consistencies. All methods associated with this model are oriented towards error prevention while still allowing the user "free-form" program input. These methods will also automatically correct certain classes of errors such as misspelled words and omitted terminal symbols of certain types and will interact with the user to gain information when there is insufficient knowledge for automatic correction.

This is part of a series of reports describing ISI research directed toward reducing significantly the cost of military software while improving its application and upgrading the general quality of software. This report covers a significant portion of the author's University of Utah doctoral dissertation, completed at ISI.

CHAPTER 1

INTRODUCTION

1.1 PROGRAMMING ENVIRONMENTS

With few exceptions, the many available programming language designs have not encouraged the development of helpful systems to make those languages more usable. Concern for the language's real users and the environment in which they must operate has been notably lacking. In fact, the user's environment has not greatly improved even with interactive computing facilities. The increasing complexity of programming tasks makes this neglect critical. Ben Wegbreit stated that his first criterion for "advanced computing applications . . . [must] be a complete ***programming system*** -- a language plus a comfortable environment for the programmer."[Wegbreit 73 : p.1] When specifically speaking on his requirements for advanced work of natural language processing, Terry Winograd also desired a cohesive programming environment [Winograd 75]. And again, in a paper describing requirements for advanced list processing, Daniel Bobrow stressed "the idea of a programming ***system*** rather than ***language***, since the programmer does not just express his algorithm, but must enter his program, test it, find bugs, modify it, etc."[Bobrow 72 : p. 2]

What do they mean by "programming environment"? It certainly must be more than on-line computing, which is already available. What is wrong with our current "advanced" facilities to make comfortable environments a major factor in future computing efforts? The problem is that most facilities are not only language-independent but also ***language-ignorant!*** We find editors which are only text editors with no built-in knowledge of programming languages. We find debuggers whose only knowledge is that of each language's common denominator -- the machine language. We find no file system based on a programming effort written in some programming language. There are some exceptions where systems do contain some of the above features; the most notable is INTERLISP [Teitelman 74], which provides a very helpful environment for the programming language LISP.

But for the most part the programmer, or group of programmers, must do a great deal more to develop a new program than simply concentrate on the semantics of the programming problem. To quote Warren Teitelman in his introduction to a paper on the predecessor to INTERLISP:

> In normal usage, the word "environment" refers to the "aggregate of social and cultural conditions that influence the life of an individual." The programmer's environment influences, to a large extent ***determines***, what sort of problems he can (and will want to) tackle, how far he can go, and how fast. If the environment is "co-operative" and "helpful" -- the anthropomorphism is deliberate -- then the programmer can be more ambitious and productive. If not,

he will spend most of his time and energy "fighting" the system, which at times seems bent on frustrating his best efforts. [Teitelman 69 : p. 1]

For example, one of the most irritating events in a programmer's activities is receiving some cryptic error message during program compilation. First, he must retreat to the text editor used to create the program, read in his file, edit it, write it out, start up the language processor again, and have it process his file (with the possibility of receiving another error message farther on). Second, while he worries about the semantics of his program, a syntax error or a semantic error detected by the compiler seems secondary to his real goal, i.e., seeing if the program operates as intended. In fact, he knows that any other programmer familiar with the programming language but unfamiliar with the particular program could usually correct any of these errors by looking at the program and the error message.

A good programming environment would alleviate much of the overhead encountered above, possibly automatically correct the error, and at a minimum provide a more agreeable atmosphere for the programming effort. A programming environment consists of a variety of tools and a system for alternating among them for different phases of program development. A *good* programming environment has "intelligent" tools and an "intelligent" system for manipulating these tools. In this context, "intelligence" means knowledge of the task, knowledge of what it is meant to manipulate, as well as application of this knowledge to aid the user in his task. The more knowledge applied, the more the tool can release the user to higher conceptual levels. In general, this applied knowledge produces the following benefits:

- Better communication and presentation of the information being manipulated.
- Better guidance provided by the tool, preventing errors and the ability of use by a less knowledgeable user.
- Better error detection and correction.
- More automated tool activities previously left to the user.

Historically there have been two classes of knowledge in programming languages: syntactic and semantic. The former has been used only for creating parsers (of course, certain semantic knowledge is also used in this task). The latter has been used in different forms of program execution. Tools currently available do not make the best use of the knowledge available.

Two good indications that a tool is utilizing its knowledge are: (1) it operates correctly and easily and (2) it helps the user in error prevention and correction (possibly automatically). Parsers have recently been developed which automatically correct errors ([Leinius 70], [Levy 71], [Peterson 72], and [Johns 74]). Also, studies in correction of the semantics of programs are emerging ([Sussman 73], [Goldstein 74], and [Wilczynski 75]). In the future we may expect new tools which apply certain knowledge for a different purpose (e.g., program verification [Deutsch 73], [Good, et al. 75], [von Henke, et al. 75], [Suzuki 75] and symbolic execution [King 75], [Hewitt, et al. 75]). The ultimate programming environment would not be static, but would be an ever-expanding system incorporating new tools as they are developed.

As previously mentioned, INTERLISP is a good programming environment. It also follows the above principle: new tools and improvements to existing ones are continuously being added. Unfortunately, this type of effort is very expensive, and is thus not feasible for every present and future programming language. Since there seem to be good arguments against the appearance of a universal programming language, research on tools for programming environments must approach the problem from another direction. This direction I would like to call LIBNI, an acronym for Language-Independent But Not Ignorant●. A LIBNI tool is a tool which knows its task for programming languages in general (or for a class of programming languages), accepts specific information about a specific programming language, and can then perform its task for that programming language.

Of course, to accomplish this, it is necessary to be able to abstract the task beyond the boundaries of one particular language. This generalization must contain forms of error protection and recovery for that particular task. Compiler-compilers, which are by definition language-independent, historically have not been widely accepted, primarily because of their poor error facilities. They produced a product that operated well on a correct program, but very badly on an incorrect one. There are two possible reasons for this: (1) their abstraction and its associated formalism would not tolerate errors and (2) even if they did, the underlying system did not utilize this tolerance well. In the case of compiler-compilers, this occurred because of the nonexistent abstraction of error recovery, which can be found in some compilers. In general, no language-independent facility can be successful if it does not include error prevention, detection, and means of correction. This is the essence of the LIBNI approach. Of course, any LIBNI tool must also incorporate good man-machine interfacing.

1.2 *PROGRAM CONSTRUCTION*

> Computer science gave us Algol 60: it also gave us the prospect of time sharing. But when we sit down at a console to write an Algol program, it is software engineering which determines how easy it is to achieve this end or alternatively, the frustrations that we have to go through. [Morton 73 : pp. 4-5]

After extensive investigation in LIBNI programming environments as a whole, the author selected the specialized area of program construction and modification. There are several reasons for this.

- This area is the most neglected and historically unrecognized as a problem area in computer science. But Edward Youngs, in a study on human errors in programming [Youngs 74], used two groups: (1) beginning programmers enrolled in their first programming course and (2) professionals, i.e., those who had been employed at least once as a full-time programmer. For the most part, both groups programmed the same tasks and used several languages: ALGOL, BASIC, COBOL, and PL/1. He categorized their errors as syntactic, logical, semantically inconsistent (e.g., undefined identifier), and clerical (e.g., a typing error). The focus in his study was the types of programming errors, not the number of errors. His statistics are relative to each group; therefore, although the

● Pronounced lib-nee.

professionals made fewer errors, this is not reflected in the statistics for their group.● For beginners, only 35 per cent of the errors were logical errors (the program *bug*, where the program runs but gives incorrect results). Thus 65 per cent were typing, syntax, and semantic conflict errors, almost all of which were caught in several passes through the appropriate compiler. The professional programmers improved by only 14 per cent. That is, approximately half of the errors made by professionals were caught by the compilers. Thus, the task of correct program construction consumes a large portion of the programming experience.

- Although designed as a part of a total programming environment, this facility is capable of being used alone. That is, it could greatly improve the phase of program construction and modification by replacing the normal text editor now being used to construct programs in the average interactive "general-purpose" computing environment.

- Unlike many other facilities, it is a component of a programming environment that, with current knowledge, could be designed in a language-independent but not ignorant manner. That is, the knowledge needed for program construction and modification can currently be abstracted.

- The errors which such a facility could prevent are irritating and unnecessary because (1) they have nothing to do with the "logical" problem being attacked and (2) they usually do not appear until later in the programming process.

Therefore, the goal was to carefully design an integrated LIBNI facility for program "editing." It should facilitate program construction by maintaining correct syntax and certain semantic consistencies (i.e., error prevention) while still allowing the user "free-form" program input. It should be able to present program structure in several forms; it should also be able to provide information concerning the use of symbols such as where and when they are defined and used. And it should react "intelligently" by giving warnings about possible inconsistencies in the use of symbols and by trying to correct simple input errors.

To accomplish this in a LIBNI manner, the facility needs a description of a particular programming language's syntax and some of its semantics related to symbols. It needs an internal representation of this information for particular programs. And it must tailor its operations based on this language description. That is, it needs three items: a language description, a representation for holding programs, and "intelligent" and adaptable operations for program manipulation.

Since these pieces are interrelated, it was necessary to design the three items together, implement a prototype to test specific parts, review the results, and then revise the design when necessary.

This design is intended to be a major part of a total environment for program development in

---

● Seven per cent of all errors in both groups were not classified for one reason or another.

the future. This system, called the Program Constructor and Modifier (abbreviated PCM), is a two-stage process. In the first stage, an "expert" creates a formal description of a programming language. This description is used in conjunction with an underlying model of programming languages to drive the second stage, in which the general user creates and modifies programs written in the particular programming language.

The PCM is important because it is the first step in creating language-independent programming environments. Not only will it be a part of the overall system, but it is one of the few parts which is also useful by itself. The PCM can be used as a separate facility, replacing the standard language-ignorant text editors used by most of today's programmers.*

1.3 *EXAMPLE SESSION*

Before the formalism and the methods for using it in creating and modifying program language structures are described, we present a scenario of an editing session. This example is not meant to give a full account of all the possible operations or modes, but only to give the reader the flavor of some tasks which can be performed and to show some of the correction facilities. This scenario is taken directly from a currently running implementation of the design.** An *expert* has previously input to the PCM the definition of PASCAL*** written in the PCM formalism with the following change made to facilitate interactive development of PASCAL programs. A PASCAL program contains global definitions, including function and procedure definitions, followed by the main program block. In this global definition phase each function and procedure definition is a separate global entity, and no ordering is implied. Also, during the session, while a reference to an undefined global entity may be made, it must be resolved by the end of the session. For example, as part of a definition of a procedure, a reference might be made to function not yet defined. This is permissible, but the function must be defined when the user declares he is finished. The liberty of this session definition has been taken because this is a system for program creation and modification, not "batch" compilation and execution.

This example will start in the middle of a user session. The user is developing a set of functions and procedures to convert assignment statements from infix to postfix notation for a simple programming language. This example was chosen for its several (but relatively short) function and procedure definitions; these make the example session short, but interesting. The reader is asked to extrapolate what he reads into more complex task domains. Although it is not necessary for the reader to understand the algorithm (since we are dealing with program

---

* The research presented here was initially inspired by the author's use of the INTERLISP system, especially the editor. Written by Warren Teitelman, the editor is a structural editor which manipulates list structures.

** A discussion of the prototype implementation can be found in Section 5.5.

*** PASCAL is an ALGOL-like language, but it does not have locally scoped variables except through the procedure and function locals. For a definition of PASCAL see K. Jensen and N. Wirth, ***Pascal -- User Manual and Report***, Lecture Notes in Computer Science, Vol. XVIII, ed. by G. Goos and J. Hartmanis (Berlin: Springer-Verlag, 1974).

structure and syntax, not semantics), a short description follows (a more complete description can be found in A.I. Forsythe, et al., *Computer Science: A First Course* (New York: John Wiley & Sons, 1969), pp. 410-417). As each lexical unit is scanned, it is associated with a number from one to six. Identifiers are associated with the value six and are output immediately. All others (i.e., operators) are assigned a value based on their precedence. A stack is used as temporary storage while the algorithm decides the appropriate order and time to output the operators.

In the example session, the user's prompting symbol for input will be "<-". Program segments input by the user will be enclosed by a matching pair of double quotes. These characters were supplied by the expert along with the language description in the PCM formalism. To avoid any confusion by a potential future user, the sequence of characters for the prompting symbol and the program segment delimiter should not appear as a meaningful sequence of characters in the programming language described by the expert. A novice PCM user might get confused as to the current state of the system if the characters typed could either be part of programming language or a prompt from the system for more input. For clarity, all user input will be typed in this character font and all output from the PCM will be in a bold-face character font. Also, user input appears only on lines with the prompting symbol "<-", immediately adjacent single-spaced lines following the prompting symbol line, and after a question has been typed by the PCM and is awaiting the programmer's answer. Between and during interactions, a continuing discussion of the abilities of the PCM will frequently appear. We will now join the user.

```
<- STATUS?

   defined globals
       procedures INFIXTOPOSTFIX, OUTPUT
       functions ENDOFSTMT, POP, STACKEMPTY, TOS
       char array STACK
       integer STACKINDEX
   undefined globals
       procedure PUSH
       function LEVEL
```

The user has just asked for a report of all globals. Notice first that the PCM has several separate definitions for the user to manipulate. The user has already input them and they are syntactically correct. It also knows the types of each global. One would usually expect to find this in a system for interactive program execution, but not in an editing system. But this is an *environment* for program construction, not a simple text editor. The user is also informed that he has not defined LEVEL and PUSH. He now proceeds to define PUSH.

```
<- DEFINE "PROCEDURE PUSH (ITEM:CHAR); BEGIN
STACKINDEX:=STACKINDEX+1; STACK[STACKINDEX] :=ITEM
END;"

   procedure PUSH defined
```

```
<- PRINT

procedure PUSH ( ITEM : CHAR ) ;
begin
   STACKINDEX := STACKINDEX + 1 ;
   STACK [ STACKINDEX ] := ITEM
end ;
```

This sequence subtly shows several PCM facilities. First, by accepting the definition, the program is syntactically correct. Second, notice that the structure of the PCM output of PUSH is different from the structure the user typed in the definition. The PCM has already discarded the text lines input by the user. The definition of PUSH is maintained in an internal form reflecting the structure of the particular programming language. Upon request, this form is converted to text lines reflecting the internal structure. Note also that the user did not have to say what to print, and until he mentions another global name he will always be referring to PUSH. The PCM maintains two pointers -- one to the current definition and one internal to the definition which is the structure of current interest. These are the same at this point, but this will change on the user's next input.

Although we did not see the user define POP, a message was printed to him at that time. POP uses STACKINDEX in the reverse order of PUSH. At the time of the definition, STACKINDEX had not been on the left hand side of an assignment statement. The user received the following message:

```
WARNING -- possible use of STACKINDEX before
it was assigned a value.
```

Notice that this is only a warning. Because it does not know the control flow during program execution, the PCM can never be absolutely sure that STACKINDEX will not have a value before it is used; at the time POP was defined a value had never been assigned, so it gave a warning. The user is now responsible for seeing that it receives a value. He may, of course, ask for a list of these warnings at any time.

He now realizes, since STACK is an array, that STACKINDEX might exceed the dimension of STACK. He decides he should change PUSH to first check STACKINDEX to see if it will exceed the dimension on the array STACK if incremented by one. If this is the case, he wants PUSH to call a system procedure ERROR, which will cause the program to abort.

```
<- FIND BEGIN

  begin <<STACKINDEX...1>> ; <<STACK...ITEM>> end
```

The PCM has found the first (and in this case the only) instance of "begin" and has changed the Current Interest Pointer (CIP) to point to the structure containing it. The PCM prints out in a short notation the structure pointed to by the CIP. This output again reflects the program structure, but in a different form. This form outputs the substructures in a limited format. The characters "<<", ". . .", and ">>" were supplied by the expert to signify the beginning, middle, and end of a substructure for this output form. The PCM uses these characters and the first and last lexeme of the substructure to create the output form. This gives the user, in a shorter and less costly way, an idea of where the CIP is currently pointing. It also gives a less than experienced programmer a better idea of the structures of the particular language. He decides to surround the statement that increments STACKINDEX with an "if" statement in order to first check its value against the dimension of STACK. But first he needs to make that statement the one pointed to by the CIP. He could, of course, type "FIND STACKINDEX", but he can also use a number indicating the position in the short output form. Each top-level element and each substructure count as one in this numbering scheme.

```
<- GO 2

  STACKINDEX := <<STACKINDEX...1>>
```

Now he is pointing to the first statement of the block, which was the second element of the short output form last typed to him. Also notice that the new substructure is the expression following the assignment operator. Here again, the structure of the language is reflected to the user. He would now like to wrap an "if" statement around it.

He could, of course, replace the assignment statement -- which is the statement pointed to by the CIP -- with an "if" statement. But since the new "if" statement will have the assignment statement as a part of it, the programmer can use the EMBED command. This command is equivalent to replacing the program segment pointed to by the CIP with a new program segment typed by the user in which the old program segment appeared as part of the new program segment. In the EMBED command, the new program segment typed by the user is searched, before parsing, for an occurrence of the special symbol "$". When it is encountered, the "$" in the new program segment is replaced by the old program segment pointed to by the CIP. Of course, the syntax of the language must be maintained. But since an assignment statement can be the else clause of an "if" statement, the user types the following:

```
<- EMBED "IF STACKINDEX=STACKLIMIT THEN ERROR ELSE $ "
```

If there were no errors in the program segment just input, the PCM would have typed the following to the user:

```
 if <<STACKINDEX...STACKLIMIT>> then ERROR else
 <<STACKINDEX...1>>
```

This shows the new program segment has been accepted by the PCM. Notice that the old assignment statement is now the else clause of the new "if" statement. But the PCM has encountered some semantic "errors" and begins the following dialogue with the programmer:

```
STACKLIMIT undefined -- is it global? YES
```

The PCM could not find STACKLIMIT in its structure. It first tried to correct a spelling or clerical error, but it did not find a close match. It then tried to further resolve the situation. It asks the user if it is global. The user responds affirmatively and the PCM proceeds. If the user had responded negatively, the PCM would then ask if the user would like to drop into a lower edit and resolve the situation himself. We will see an example of this in a slightly different situation below. The PCM continues to process the EMBED command.

```
system procedure ERROR requires one parameter
do you want to edit? YES
    <<ERROR>>
```

The PCM finds a semantic error. It is context-dependent on ERROR having one parameter. This information about system procedures was again supplied by the expert. The PCM then asks the user if he wants to edit to correct this error. He does, and the PCM prints the program segment at the point of the error, which because of the error is also set as the new CIP.

```
<-- PARAMETERS? ERROR

   first parameter is MESSAGE : char

<-- EMBED "$ ('stack overflow')"

       <<ERROR...)>>
```

Notice that the herald has changed to "<--", which indicates to the user that he is in a lower edit and that the initial error is suspended waiting for this edit to return. When it does, it will check to see if the error has been resolved. The user then asks for the arguments (parameters) to ERROR and then edits the structure to add the error message.

```
<-- DONE

   OK
```

The user returned and the PCM was satisfied. The PCM now continues and finally succeeds in executing the original EMBED command.

```
   if <<STACKINDEX...STACKLIMIT>> then <<ERROR...)>> else
   <<STACKINDEX...1>>

<- TOP

<- PRINT

procedure PUSH ( ITEM : CHAR ) ;
begin
   if STACKINDEX = STACKLIMIT
       then ERROR ( 'stack overflow' )
       else STACKINDEX := STACKINDEX + 1 ;
   STACK [ STACKINDEX ] := ITEM
end ;
```

The user resets the CIP to the beginning of the procedure via the "TOP" command. Then he asked to see the entire procedure definition. He then remembers that STACKLIMIT is still undefined.

```
<- DEFINE "var STACKLIMIT:INTEGER;"

   var STACKLIMIT defined

<- UNDEFINED?

   function LEVEL
```

The only undefined global referenced by some other procedure or function is LEVEL, which is the function that assigns a precedence number for all lexical units.

```
<- DEFINE "function LEVEL(LEXEME:CHAR) : INTEGER;
begin case LEXEME of
'↑' : LEVEL:=5;
'*', '/' : LEVEL:=4;
'+', '-' : LEVEL:=3;
'<', '=', '>' : LEVEL:=2;
'(', '←' : LEVEL:=1;
')' : LEVEL:=0
end end;"

   function LEVEL defined
```

```
<- PRINT

function LEVEL ( LEXEME : CHAR ) : INTEGER ;
begin
  case LEXEME of
      '↑' : LEVEL := 5 ;
      '*' , '/' : LEVEL := 4 ;
      '+' , '-' : LEVEL := 3 ;
      '<' , '=' , '>' : LEVEL := 2 ;
      '(' , '←' : LEVEL := 1 ;
      ')' : LEVEL := 0
  end
end ;
```

In the semantics of the language PASCAL the value of a function is the last value assigned to the name of the function before it returns. The user now has a function which will return a number for each of the allowed operators plus the parentheses. It still needs to return a number for variables.

```
<- FIND begin

  begin <<case...end>> end

<- INSERT BEFORE 2 "LEVL :=6"

  syntax error
  adding ";" to input after ":= 6" -- OK? YES

  LEVL undefined -- is it LEVEL? YES

  begin <<LEVEL...6>> ; <<case...end>> end

<- PRINT

begin
  LEVEL := 6 ;
  case LEXEME of
      '↑' : LEVEL := 5 ;
      '*' , '/' : LEVEL := 4 ;
      '+' , '-' : LEVEL := 3 ;
      '<' , '=' , '>' : LEVEL := 2 ;
      '(' , '←' : LEVEL := 1 ;
      ')' : LEVEL := 0
  end
end
```

Here we see the first syntax error. The parse is suspended and the syntax error corrector finds that the parse could continue if a semicolon were added to end the statement. It asks the user, who responds affirmatively. The parse continues and succeeds. But then a "semantic" error is encountered. LEVL could not be found (similar to STACKLIMIT above), but a close spelling match was found with LEVEL. The PCM continues and prints out the result of the above addition.

Now there are no undefined globals. The user does want to check, however, to see if there are any routines that use STACK inadvertently (i.e., not a stack manipulation function).

```
<- USES? STACK

  STACK used in POP, PUSH, STACKEMPTY, TOS
```

The user is now satisfied that STACK is used only by the appropriate routines. At this point the user could do several things, depending on what other facilities were available to him. The next goal of this research would be to incorporate into the same environment an interpreter/debugger/compiler system for evaluation and logical debugging in which the debugger communicates with the user in a style compatible with the programming language and in its terms and structures. Until this goal of being able to execute and debug the programs in the same environment is reached, the user could write out his programs onto the file system and then have them compiled by the appropriate compiler. Even this could greatly help the user. By knowing that his programs are syntactically correct and his symbol table information is correct, he gains a great deal of confidence about the success of the compilation on the first try, and can then concentrate on his logical errors. The ultimate goal of this research would be an ever-expanding environment encompassing all helpful facilities as they are developed (as initially discussed in Section 1.1).

1.4 SYSTEM OVERVIEW

Although the example session dealt with PASCAL, the PCM is language-independent. A definition of PASCAL had been previously input and the PCM used this definition to take the appropriate actions relative to the particular programming language. The language definition is given in a formalism for describing syntactic information and certain semantic information about the use of names in the language. This description is then used to build the internal structure for a particular statement in the language. It is also used to decide which actions are "legal" for the user to instigate and for error prevention and correction. That is, there are three major components of the PCM:

1. The formalism for describing, by a language expert, the syntax and certain semantics of programming languages.

2. The structure used to hold the program in internal form and maintain information concerning the use of names.

3. The methods -- including the parser -- which use the formalism as a guide in the manipulation of the structure.

Each of these will be discussed in detail in forthcoming chapters, but a short informal discussion is presented here to give the reader an overview of the entire PCM.

The PCM Language Formalism

This newly designed formalism is the key to the LIBNI aspects of the PCM. All methods -- used here to mean any action taken by the PCM -- are based on the formalism. That is, each action has different constraints and possibly different results based on the kind of mechanism in the formalism used to describe a particular program construct.

This formalism is a production language* whose left-hand side is a generic term of the grammar. The right-hand side is made up of other generic terms (i.e., nonterminals), terminals, and five metaconstructs, which are descriptions of how to determine the positioning and occurrences of generic terms and terminals within their scope. Their use will be made clear by their enumeration.

1. Ordered Sequence -- A linear sequence of items in the grammar. For example, an assignment statement in a particular programming language may be an identifier, followed by the assignment operator, say ":=", followed by an expression. This is an Ordered Sequence with two nonterminals -- identifier and expression -- and one terminal -- ":=".

2. Alternative Set -- A set of items in the grammar, of which only one can appear at that particular point. A nonterminal called "operator" might have as its production the Alternative Set containing "+", "-", "*", "/", and "↑".

3. Alternating Sequence -- A metaconstruct reflecting a finite but unlimited number of alternations of two items. It also requires that this sequence start and end with an instance of the first item. For example, a set of statements may be an Alternating Sequence with the first item being the generic term statement and the second item being the terminal semicolon.

4. Bracketed Sequence -- A sequence of three items in the grammar, of which the first and last items must be terminals and the middle item can be of any form. This can be seen in parameter lists where the first and last items are "(" and ")" respectively and the middle item is an Alternating Sequence of expressions and commas.

5. Repeating Sequence -- An unlimited number of repetitions of one item in the

* Terminology used in this paper is in common usage when discussing formal language descriptions (e.g., terminal, nonterminal, productions, etc.). For a discussion of these terms see John E. Hopcroft and Jeffrey D. Ullman, *Formal Languages and Their Relation to Automata* (Reading, Mass.: Addison-Wesley Publishing Company, 1969).

grammar. For example, an identifier in ALGOL 60 would be described in this formalism as an Ordered Sequence of an alphabetic character followed by a Repeating Sequence of an Alternative Set of alphabetics and numeric characters.

This set of metaconstructs gives the expert a flexible and natural way to express the grammatical structures of programming languages.

There is one other construct in the formalism, but it does not describe any syntax. Instead, it is a declarative statement about the use of names in the language. If a production rule succeeds (i.e., matches some input), this statement becomes reflected in the structure. For example, in an ALGOL 60 program, after the successful parsing of a declaration statement, the variable is now defined and has an associated data type. This will be reflected in the structure which holds that particular program segment.

As was stated previously, actions taken by the PCM use as their basis for decisionmaking the language description written in this formalism. It also uses it as a template for the structure which internally holds the program and its associated information.

The Structure for Holding Programs

The PCM structure is the data base which reflects the dynamic state of program construction. For a single program, it is a highly augmented *n*-ary parse tree of that program. It is also the "organizer" or "file system" for a set of programs. This structure, which represents the user's PCM activities, is called PCMREP. It differs from a normal parse tree in several ways.

First, all terminals are maintained in the tree. That is, if a program traversed PCMREP from left to right and top down (i.e., preorder) and output every leaf node, the user would see every character he had input, including "grouping" terminals such as parentheses. There are three reasons for this. First, the structure can easily be unparsed for presentation to the user. Second, any change in the program is a change in PCMREP and is immediately reflected to the user. Third, the programmer will not lose the ability to use as a visual aid groupings (such as parentheses) that do not cause a change in the parsing.

Next, each segment in PCMREP which represents a production rule in the formalism has a reference to that rule and the alternative, if there was one, on the right-hand side of that rule which caused that rule to succeed. This information is used by the PCM for certain interactions with the parsing mechanism. In the example session, when the user wanted to insert the assignment statement before the case statement, the PCM looked at the production for a block, saw that the only legitimate possibility for an insertion there was an Alternating Sequence of statements and semicolons, i.e., a block is a "begin" followed by alternating statements and semicolons and finished by an "end", therefore the user could only be inserting statements and semicolons. The PCM started the parser on that Alternating Sequence. When the parser finished, the PCM added the new segment to the existing PCMREP, thus saving any unnecessary unparsing and reparsing of the entire program structure. This information is also used by the PCM with no parsing interaction. For example, if the programmer wanted to delete some segment, the PCM would first check to make sure that the deletion did not cause a syntactic discrepancy in the more global structure.

The third difference is a special node in PCMREP called the *access node*. Access nodes appear in the structure whenever a new naming context would be created during program compilation. Access nodes are extended symbol tables for every symbol used in the tree below it. Each entry has a series of attributes which describe not only its data type, but also information concerning where it is used, where it is set, where it is created (declared), how to access the parent node if it was not created in the structure below, and possibly a syntactic pattern which must follow it (e.g., number of parameters). This information is used by the PCM to determine semantic consistencies and to provide information to the user. We saw several occurrences of this in the example session.

Although the formalism and the structure are both integral parts of the PCM, they are passive in that they are manipulated but not manipulating. The active part of the PCM is in the facilities which establish a symbiosis between a programmer and the computer in working toward the goal of correct program construction.

The PCM Activities

The PCM activity can be divided into two major sections: (1) the parsing technique, which includes error correction, and (2) the manipulation and presentation of the structure and error prevention.

The parsing mechanism builds the appropriate substructure for insertion into PCMREP and also takes care of the syntactic and semantic errors. An important part of the correction facilities lies in the concept of the user as a process, which was first discussed by Alan Kay and James Mitchell in their dissertations ([Kay 69] and [Mitchell 70]). During correction the user can be invoked to answer questions and take actions. The PCM does not have to make erroneous corrections or assumptions, since the user is always available for advice.

The parsing mechanism operates in two distinct phases, both with their own correctors (see Figure 1). First the input is parsed, which might involve the syntax corrector and the user. By the time this process is complete, the structure is syntactically correct and, except for the semantics, is ready to be inserted into the program structure. During the parse, the semantic statements have been built up and are now ready to be processed by the semantic checker. If the checker finds an error or discrepancy, it invokes the semantic corrector, which may again interact with the user. When this process is complete, the PCM knows that whatever modification was made to the structure is correct.

The manipulation functions first analyze the request in the current context on the basis of rules based on the formalism. If it is necessary to invoke the parsing mechanism, each function initiates it with an appropriate production rule. For example, at some points in PCMREP a delete of a particular node is not valid because it would cause a syntactic inconsistency. Before any modification is really made to the structure, the delete function will make sure that it is a valid operation.

The presentation functions are divided into two modes: the long form ("pretty print") and the short form ("structure print"). Both are based on the metaconstructs of the formalism. For example, in long-form-mode processing of a Bracketed Sequence, it is first determined if all of the text can be output on the remainder of the current line. If not, and if the start and finish

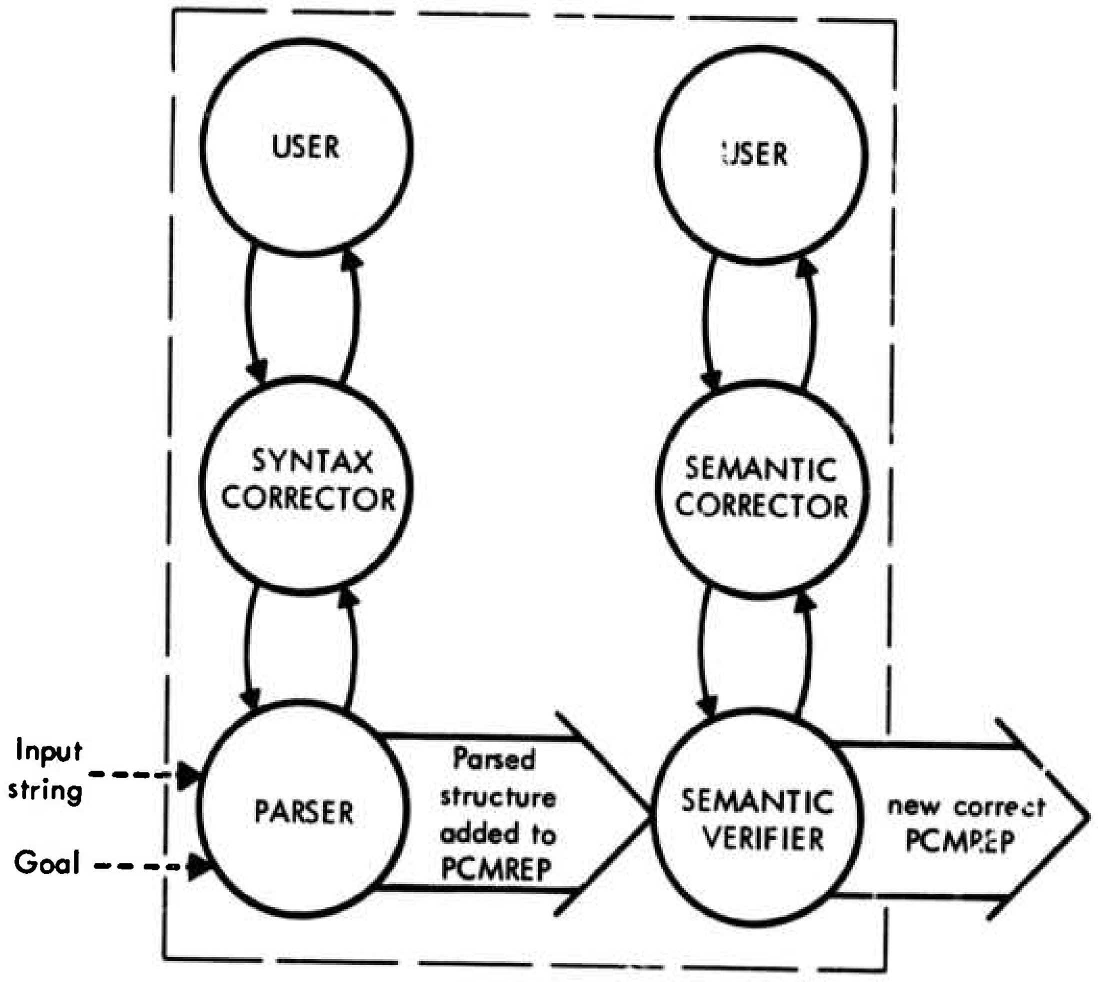

Figure 1. PCM's parsing mechanism

terminals are alphabetic, then the start terminal is output on one line, and the middle form is output using its own rules, followed by the end terminal on its own line. Of course, all this is done with the appropriate indentation.

The PCM system, as overviewed above, will be described in detail in Chapters 3, 4, and 5. Chapter 2 discusses the previous research in areas coinciding with parts of the PCM and makes comparisons between the former and the latter. Conclusions and suggestions for future research will be given in Chapter 6.

CHAPTER 2

ASSOCIATED RESEARCH

This research focuses on language independent tools for programming environments. Although there are many such tools -- editors, debuggers, etc. -- most are ignorant of the particular syntactic and semantic structures of the languages with which they are used. Below we deal primarily with environment tools which are language-independent but adaptable to deal with particular languages. The language-independent aspect of other research will be stressed because a generalization of the appropriate concepts was necessary to raise them to a state of language independence. This chapter mainly deals with language editors, presentation facilities, and error-correcting parsers.

2.1 LANGUAGE EDITORS

Language editors can be divided into two categories: language-dependent and language-independent. The former are editors written for a particular programming language. Both categories of editors are in some form of interactive system. For the most part, only editors which edit the structures in the language are of interest.

Language-Dependent Editors

Several interactive systems have built-in text editors (e.g., APL, JOSS, JOVIAL), but the user edits the text string representation of the program. An extension of this was provided for the JOVIAL system [Bratman, et al. 68], where each line was checked for syntax errors after the editing of that line. It also had some primitive globals checks -- for example, that there be the same number of BEGINs and ENDs. More recently, in COPILOT, a system for a multiple processing environment, a CRT-oriented text editor was provided and reparsing was accomplished just prior to program execution [Swinehart 74].

The INTERLISP editor is a language editor which uses as its basis for editing the language structure and not text. LISP has only one structure for programs -- a binary tree (and, of course, atoms); a canonical list representation is defined in terms of these primitives. The editor has facilities for moving, deleting, and adding list structures. It also provides several ways to move through and present the lists. Also, since the program structure is identical to the data structure, a user may use the same editor to edit his data structures. In fact, Robert Balzer [Balzer 74], an INTERLISP user, considered it such a good programming environment that he, as an experiment, tried to use it as a front-end to the ECL programming system.●

● ECL is another interactive programming system. For a description of the language and system see Wegbreit 71.

Besidesd taking advantage of several environment features, he also used the INTERLISP editor by putting parentheses around ECL statements and then had INTERLISP read it as a list structure which could then be edited by INTERLISP's structured editor. Unfortunately, this changed ECL's syntax.

Because of the dynamic properties of LISP, the only intelligence the editor can exhibit is to insure that the program is syntactically a correct list structure. For the same reason, this is all the PCM could accomplish with LISP, if it were tried. But because INTERLISP also has an execution environment, Warren Teitelman has provided a run-time error correction facility which includes spelling correction and parentheses mismatch correction.

Although the PCM can operate with a definition of the language LISP, it is more oriented to less dynamic programming languages: languages which have static scope rules and, preferably, declared variables. This allows the PCM to make static semantic analyses, which must be put off until program execution time in languages similar to LISP.

Language-Independent Editors

Little research has been done in this area but there are two systems. The first, done by Wilfred J. Hansen, called EMILY [Hansen 71], uses a Backus-Naur Form (BNF) for the language description formalism. The user sits at a graphical display equipped with a lightpen. Starting from the initial BNF rule, the possible alternatives are presented to the user. Using the lighpen he selects the desired alternative, and the alternatives for that rule are presented. He repeats this process until he reaches a rule which needs a user-input lexeme (i.e., a variable name or constant from the keyboard). The user never inputs constant terminals of the language, such as key words, operators, parentheses, etc. He selects the alternative which produces those terminals. Since the user is driven by the BNF description, he can never input incorrect syntax. All the user can do is select alternative rules and input identifier, string, and numeric constants. This may appear at first glance to be an excellent system for program input, but even Hansen agrees that except for its possible use as a pedagogical tool for teaching a new programming language syntax it is too tedious to use as a creative tool. One example, in his report, shows the user at the BNF rule <ASGN STMT>● and he desires to input

S = S + A(I) ;

This requires eleven selections of alternatives using the lightpen and four interspersed keyboard entries for the variables, for a total of fifteen separate interactions with the system. Even a novice programmer would quickly find this excessive. He does, however, provide a simple symbol table which allows the user to iteratively view each program fragment in which that symbol appears.

A similar system, in fact based on the above research, was done by David Lasker, but it is not entirely language-independent [Lasker 74]. He input the definition of the programming language SUE [Atwood, et al. 71] by its BNF description, but he added an "escape" alternative

---

● The BNF describes a subset of PL/1 and this rule describes the assignment statement for that language.

for the rule <EXPRESSION> which invoked a parser for expressions in SUE. He also added a user-invoked function for discovering undeclared identifiers. Both of these added features were written dependent on the definition of SUE.

2.2 PROGRAM PRESENTATION FACILITIES

Most tools which format programs are language-dependent. STYLE, a program for formatting ANSI-FORTRAN programs [Lang 72], is a batch-run system which rearranges programs using a standard algorithm. But it can be directed by parameters supplied by the user specifying such formats as the amount of indentation for a DO loop. It also does a minimal amount of syntax checking.

FORTRAN has a very limited structure; thus, more interesting are the facilities for formatting programs found in most LISP systems -- a highly structured language. These automatically construct text output which reflect the structure and nesting of internal lists. Ira Goldstein describes several algorithms with different rates of success for this.function [Goldstein 73].

The EMILY-based systems, the only language-independent systems which format programs, are directed by the expert who wrote the BNF description of the programming language. The BNF contains formatting instructions such as indenting, when to go to the next line, etc. That is, the EMILY systems do no computation based on the line length, the program structure, etc., and are dependent on the expert to determine the correct output format.

2.3 ERROR-CORRECTING PARSERS

Language-independent parsing techniques which have a considerable degree of error correction have been appearing in the literature in the last several years.● They assume that the input is reasonably close to being a correct program and that the fewest changes made to the input will yield the "best" results. This is referred to as *minimum distance correction*, where distance is measured as the difference between the original input string and the modified string. This is formally treated in [Peterson 72]. He gives an algorithm for this form of correction; unfortunately, it is very slow: it is proportionate in time to the cube of the length of the input string. It was argued that this is unrealistic for compilers, and studies were made to decrease this to a time linearly proportionate to the length of the input string ([Levy 71] and [Peterson 72]). These are not minimum distance algorithms, but have heuristics trying to approach the results of minimum distance correction. The above papers treat this as a formal language problem and report no actual implementations . Other research was done with running parsers using different heuristics and the results have been excellent ([LaFrance 71], [James 72], and [Johns 74]), that is, usually better when compared with language-dependent parsers with their own built-in error correction.

All of these programs were designed for "batch" compilation and thus cannot receive advice from the creator of the program. And since the corrections are not necessarily the "correct" correction, they sometimes cause more errors farther along in the parse. These are referred

● The earliest work appeared in 1963 by E.T. Irons.

to as avalanche errors. When an avalanche is detected, the better programs try to recover by discarding elements of the input string until a safe delimiter appears (often a semicolon) that will let them continue parsing.

The PCM, on the other hand, is an interactive system in which the user plays a vital role. The user is consulted on corrections and he can, in certain cases, edit the input string either by direct intervention or by the corrector exceeding constraints of time or number of changes.

Of course, other research can be indirectly associated as influential on this research effort (e.g., NLS [Engelbart 70]); when particularly relevant, it will be mentioned briefly in the text. The remainder of this report will give the reader a detailed accounting of the PCM design followed by a summary and suggestion for further study.

CHAPTER 3

THE FORMALISM FOR LANGUAGE DEFINITION

The new formalism presented here is the mechanism whereby the expert describes a particular programming language to the PCM. This description drives not only the parsing mechanism, but also the information base whereby the PCM activities determine the legitimacy of a particular operation on some portion of PCMREP. These activities will be described in Chapter 5.

The formalism is a production system which contains a sequence of statements of the form

nonterminal -> definition

Each definition is (1) another nonterminal, (2) a terminal, or (3) one of the metaconstructs. Each metaconstruct contains elements which can also be nonterminals, terminals, and other metaconstructs.

Before the metaconstructs are presented, a few notational considerations must be discussed. Each element of a definition is called a *form* of the grammar and is designated in the notation as f.n where n is a number or letter used as a subscript. Ellipses are used in the notation and examples to designate missing forms; they are not elements of the formalism. Terminals in the grammar are enclosed in single quotes and nonterminals are lower case character strings (not including space) not enclosed in quotes. Where terminals are necessary in the notation, they will be designated as t.n (a subset of f.n). Example uses of the metaconstructs will be taken from several high-level programming languages.

3.1 THE METACONSTRUCTS

Ordered Sequence

Notation: [f.1 f.2 . . . f.n]

An Ordered Sequence describes a linear sequence of forms in the grammar. For this metaconstruct to match some input, i.e., succeed, an instance of each form must be recognized in the order they appear in the notation.

Examples:

```
assign-stmt -> [ identifier ':=' expression ]

while-stmt -> [ 'WHILE' boolean-expression 'DO' statement ]

if-stmt -> [ 'IF' boolean-expression 'THEN' statement
     'ELSE' statement ]
```

Alternative Set
Notation: { f.1 | f.2 | . . . | f.n }

An Alternative Set is a set of forms of which one form must match the input. The first form matched causes the metaconstruct to succeed, and no further matching is tried. There is a special form, EMPTY, which causes an immediate succeed with no input being recognized. This should, of course, be placed at the end of the set.●
Examples:

```
statement -> { assign-stmt | while-stmt | . . . | if-stmt }

labled-stmt -> [ { [ label ':' ] | EMPTY } statement ]
```

and this extended definition from the previous examples

```
if-stmt -> [ 'IF' boolean-expression 'THEN' statement
     { [ 'ELSE' statement ] | EMPTY } ]
```

Alternating Sequence
Notation: < f.1 f.2 >

An Alternating Sequence represents finite but unlimited instances of alternations of the two forms. This sequence must begin with an instance of f.1 and end with an instance of f.1. This sequence may, at a minimum, be a single instance of f.1.
Examples:

```
block-body -> < statement ';' >

declaration-stmt -> [ 'DECLARE' < identifier ',' > data-type ]
```

and this production without precedence

```
arith-expression -> < identifier { '+' | '*' | . . . | '↑' } >
```

● This is not a definition issue (for that it could be placed anywhere in the Alternative Set), but rather an issue for the recognizer which is driven by a left-to-right try of each form.

Bracketed Sequence
Notation: [) t.1 f.1 t.2 (]

The Bracketed Sequence is a special form of the Ordered Sequence. It is a three-element sequence with the terminals of the grammar (t.1 and t.2) surrounding the inner form (f.1). This metaconstruct is provided since it occurs often and aids in error prevention.
Examples:

```
parameter-list -> [) '(' < expression ',' > ')' (]

block -> [) 'BEGIN' block-body 'END' (]

arith-expression -> < { identifier | [) '(' arith-expression ')' (] }
        { '+' | '*' | . . . | '↑' } >
```

Repeating Sequence
Notation: [* f.1 *]

A Repeating Sequence represents a sequence of one or more occurrences of the form enclosed.
Examples:

```
block -> [) 'BEGIN' { [* [ declaration-stmt ';' ] *] | EMPTY }
        < statement ';' > 'END' (]

identifier -> [ letter { [* { letter | digit } *] | EMPTY } ]
```

This ends the syntactical portion of the formalism. The semantic portion of the formalism will be presented more easily after the structure is discussed. It will be described in Section 5.4, which discusses the semantic checker.

3.2 WHY NOT BNF?

A legitimate question might be raised as to the necessity of a new syntactic formalism -- "Why not BNF?" There are three major reasons for not using BNF or one of its derivatives. The first, which has been argued elsewhere, is that BNF forces excessive structure when the grammar has rules for an indeterminate number of repetitions of a sequence. BNF forces the use of recursion to generate a deep structure when a linear structure is really the intuitively desired one. Example:

```
<parameter list> ::= <expression> , <parameter list> | <expression>
```

This construct will generate the structure in Figure 2, but the desired structure can be expressed in the PCM formalism as

```
parameter-list -> <expression ',' >
```

and generates the linear structure in Figure 3.

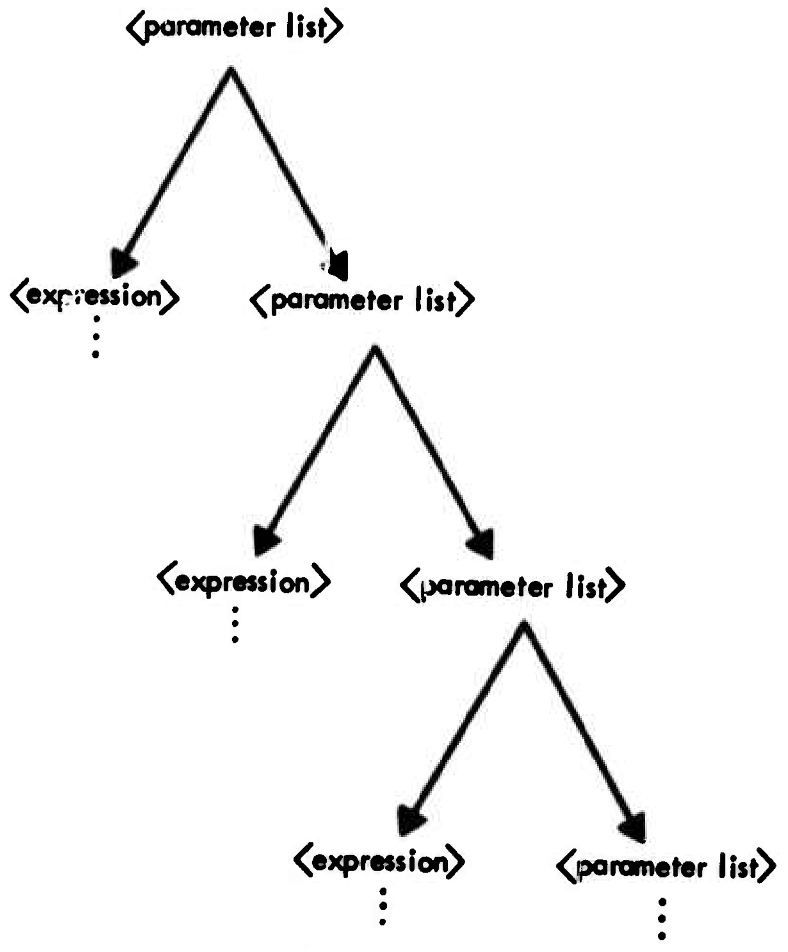

Figure 2. The Tree for

⟨parameter list⟩ ::= ⟨expression⟩ , ⟨parameter list⟩ | ⟨expression⟩

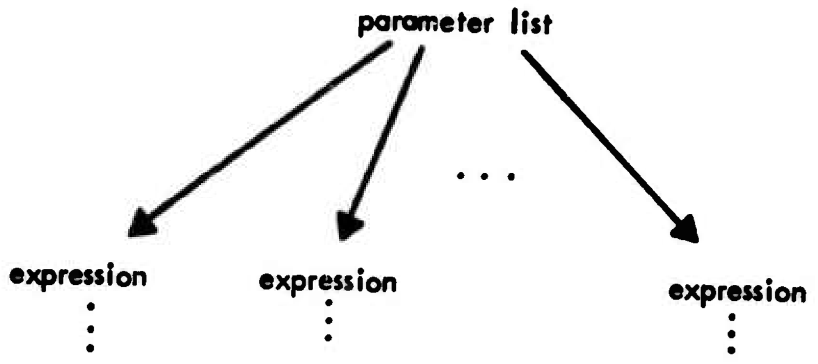

Figure 3. The Tree for

parameter-list -> < expression ',' >

Not only does the BNF cause the intuitively wrong structure, but it also generates excessive nonterminals (i.e., <parameter-list>). This is counter-intuitive. The expert may even decide to linearize constructs with precedences, such as arithmetic expressions, for ease in editing. Also, the linear structure is more easily converted to text for presentation to the programmer.

The second reason is related to the first, but it also covers a broader spectrum. The PCM formalism gives the expert the ability to describe language constructs straightforwardly while BNF forces the expert to describe his language indirectly. This is reflected in the excessive number of nonterminals for a particular language. A simple example will serve as an illustration.

```
<block> ::= BEGIN <block body> END
<block body> ::= <statement> ; <block body> | <statement>

                          versus

block -> [) 'BEGIN' < statement ';' > 'END' (]
```

This concise method of describing a language syntax can be further illustrated through the syntactic description of PASCAL through the statement level (i.e., from program to block to statement). The BNF description of just the PASCAL language construct, "statement", as found in [Jensen 74] uses twenty-two nonterminals (productions) to describe the same syntax as the single nonterminal, "statement", found in the following:

```
program -> [ block '.' ]

block ->
  [   [*  {   [) 'LABEL' < unsigned-integer ',' > ';' (] |
              [ 'CONST' [* [ identifier '=' constant ';' ] *] ] |
              [ 'TYPE' [* [ identifier '=' type ';' ] *] ] |
              [ 'VAR' [* [ < identifier ',' > ':' type ';' ] *] ] |
              [   {   [ 'PROCEDURE' identifier parameter-list ] |
                      ['FUNCTION' identifier parameter-list ':' type-identifier ] }
                  [) ';' block ';' (] } *]
      [) 'BEGIN' < statement ';' > 'END' (] ]

statement ->
  [   {   [ unsigned-integer ':' ] | EMPTY }
      {   [ { variable | function-identifier } ':=' expression ] |
          [ procedure-identifier { [) '(' < expression ',' > ')' (] | EMPTY } ] |
          [) 'BEGIN' < statement ';' > 'END' (] |
          [ 'IF' expression 'THEN' statement { [ 'ELSE' statement ] | EMPTY } ] |
          [ 'CASE' expression 'OF' < [ < constant ',' > ':' statement ] ';' > 'END' ] |
          [ 'WHILE' expression 'DO' statement ] |
          [ 'REPEAT' < statement ';' > 'UNTIL' expression ] |
          [ 'FOR' variable-identifier ':=' expression { 'TO' | 'DOWNTO' }
              expression 'DO' statement ] |
          [ 'WITH' <variable ',' > 'DO' statement ] |
          [ 'GOTO' unsigned-integer ] } ]
```

THE FORMALISM FOR LANGUAGE DEFINITION

The third and most important reason for a syntactic description using multiple metaconstructs is to provide the PCM a high-level view of the syntax. The PCM operations vary depending on which metaconstruct generated the structure being manipulated. For example, an insertion into the structure is valid for an Alternating Sequence, since there may be any number of alternations. But an insertion is valid in an Ordered Sequence only if the position of the insertion is the EMPTY alternative of an Alternative Set within the Ordered Sequence. An example of this may be found in the PASCAL description above, where the description of the PASCAL "if" statement has as its fifth form either an else-clause or EMPTY. If initially the PCM user had by default used the EMPTY alternative, he could then later insert an else-clause.

An analogy on the semantic level can be made. Consider a programmer (the PCM) trying to change another programmer's work. If the program was written in a high-level programming language (the PCM formalism) rather than assembler language (BNF), then there is a better chance the programmer will more easily see the intent and structure of the program and also be more confident that the changes he makes are valid.

This formalism is used as the template for the structure which will hold the programs being constructed. This structure (PCMREP) is presented next, followed by a description of the methods used to modify, construct, and present this structure to the user.

CHAPTER 4

THE STRUCTURE FOR HOLDING PROGRAMS

The PCM structure (PCMREP) is the data base which functions as a file system for a programming effort and a receptacle for parsed programs. Although it is similar to a normal parse tree, considerable extra information is associated with the nodes to facilitate the various editing and display operations. In the discussion below, the differences from standard parse trees will be emphasised.

4.1 THE PARSED STRUCTURE

First, all terminal symbols are maintained in the structure; this includes operators, parentheses, key words, etc. as seen in Figure 4. There are several reasons for this.

- The structure can easily be unparsed for presentation to the user. There is no need to compute whether "grouping" terminals are necessary during the unparse.

- There is no need to maintain a separate string representation of a program which needs to be updated whenever a modification takes place. Any change in the program is a change in PCMREP and is immediately reflected to the user.

- The programmer will always see the program as it was input, except for the structuring, of course. For example, in a logical expression he may use parentheses which do not change the result of the parse but which are used by the programmer as a visual aid for later review.

Also, each nonterminal in PCMREP contains some associated information about the parse. It contains not only which rule in the formalism caused the successful completion of that part of the parse, but also which alternative of that rule. This information is used by the PCM to determine legitimate manipulations. For example, in Figure 4 the uppermost statement nonterminal has a pointer to

['IF' expression 'THEN' statement { ['ELSE' statement] | EMPTY }]

If the programmer requested to insert some piece of program after element four, the PCM would see the Alternative Set

{ ['ELSE' statement] | EMPTY }

after element four and see that EMPTY was the initial alternative chosen and start the parsing mechanism using as its goal

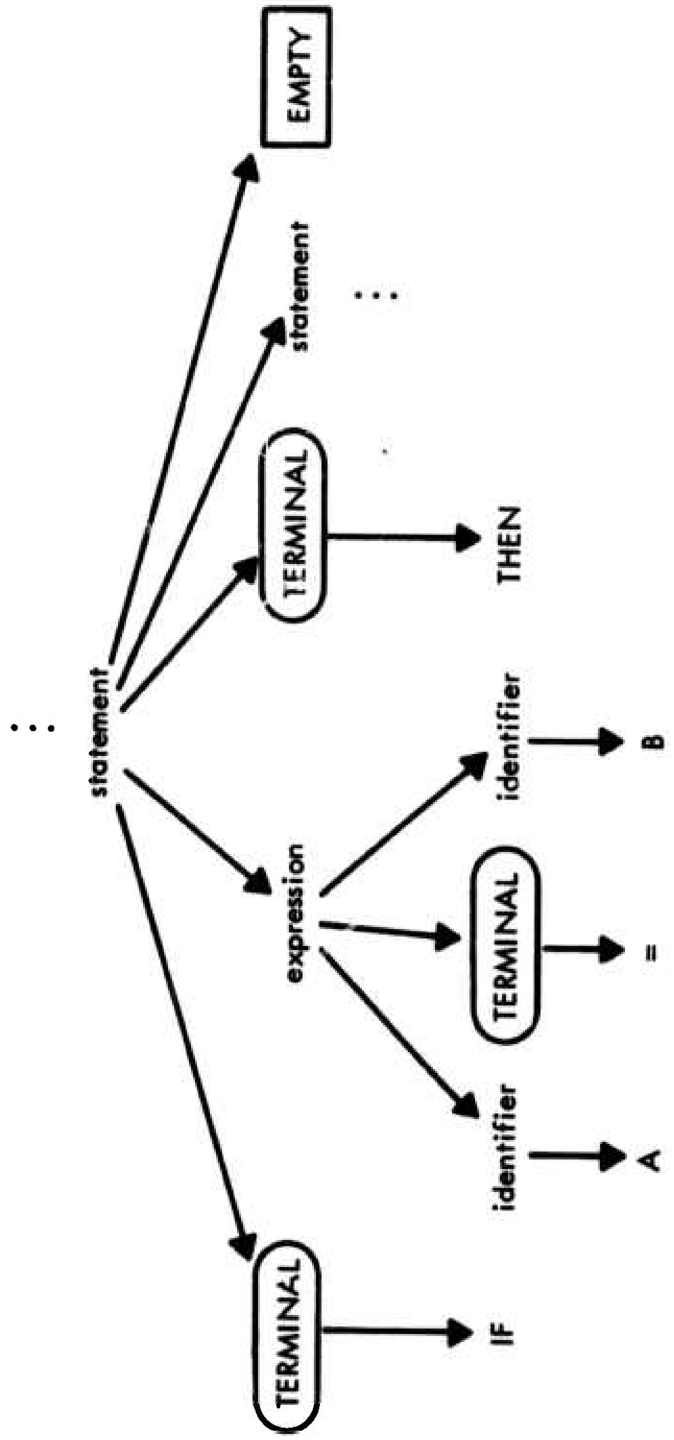

Figure 4. A PCMREP segment

goal -> { ['ELSE' statement] | EMPTY }

Or, alternatively, if there was already an else-clause, the PCM would not allow the user that modification, but it would allow the deletion of the else-clause without invoking the parsing mechanism at all by replacing it with EMPTY. This saves any unnecessary unparsing and reparsing.

4.2 ACCESS NODES

Access nodes are special information nodes which maintain the correct relationship between symbols. They appear in PCMREP whenever a new naming context would be created during program compilation. This node contains information concerning *every* symbol to which the program would have access. The PCM maintains two access nodes -- system and global -- which reside at the top of PCMREP. Each access node maintains two sets of information: a pointer to the next access node more global in scope and a symbol table of all symbols which are referenced in the scope of this access node. This structure can be seen in Figure 5. Here we see a global procedure FOO (each box in this figure with a name in it represents an element of the symbol table for that access node). Each entry in the symbol table contains a considerable amount of information concerning the use of that symbol in the structure below it. We will see the exact information later in this section.
We are now concerned with the access nodes. Notice that FOO could be moved to another place in PCMREP by just changing its definition pointer and context link. By looking at the structure we see that X is local to the leftmost access node. And, although it cannot be determined if both Y and Z in the rightmost access node are used there but declared in FOO, it appears to be the case since they appear in both associated access nodes. That is, they were probably declared at the top level of FOO and used in the rightmost program segment. To make sure we would have to look at the attributes of these entries in the symbol table information in the lower node to see if they were just used there or also declared there.

The Symbol Table

The symbol table is as an association list of *symbol:value* pairs, where *value* is itself an association list of *attribute:value* pairs. Several of these *attributes*, besides being information holders, also act as "switches". That is, if the *attribute* does not have a *value*, then the switch is false. The list of possible attributes of a symbol are

Data-type, whose value is any element of the set of possible data types for the particular programming language. This information is mainly used by the spelling correction mechanism to help find the most likely candidates for a "close" match.

Declared is a boolean flag which states whether the symbol has been declared or not.

Occurrence-dependency-set, whose value is a set of pointers into the lower structure. These point to statements that depend on the existence and attributes of the symbol to maintain the validity of the statement. This is used in conjunction with the next attribute for quick evaluation of the impact of a change to a declaration statement.

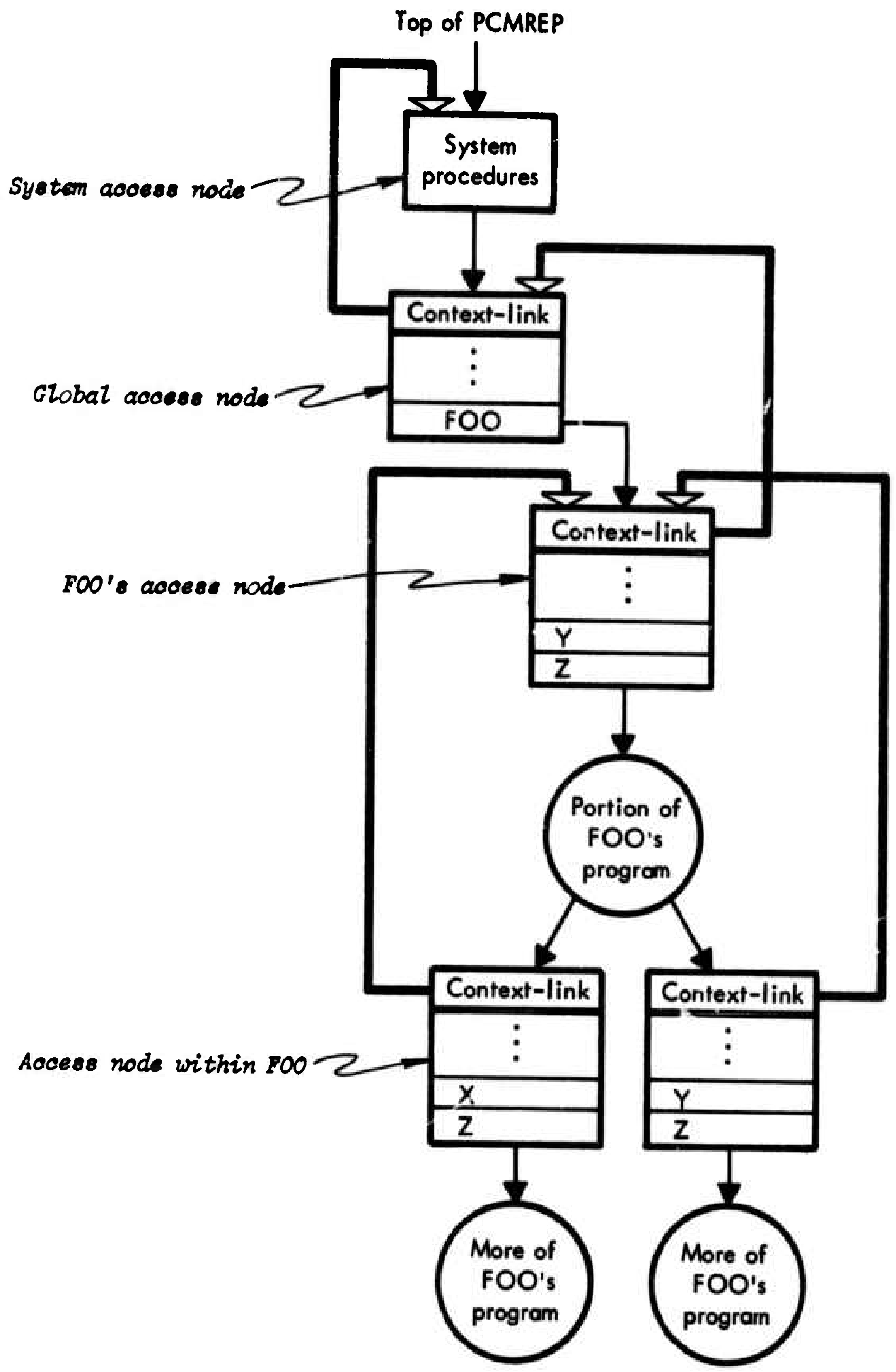

Figure 5. An example of access nodes in PCMREP

Declaration-pointer points to the statement that caused the initial creation of this symbol. If this attribute does not have a value, the declaration was made at a higher level in the structure.

Context-mode describes the way to traverse up the structure to arrive at the access node which contains the *declaration-pointer* for this symbol.

Set-flag-dependency-set is a set of pointers into the lower structure similar to the ***occurrence-dependency-set***, but it contains only pointers to those statements which assign to or pass as a reference this symbol. If this set is NULL, this fact is used to determine a possible error condition. Whenever an entry is being added to the ***occurrence-dependency-set*** and the ***set-flag-dependency-set*** is NULL, the PCM informs the programmer of the possible problem that he might be using the symbol before it was assigned a value. Because of the unknown control flow of the actual program execution, the PCM cannot be positive that the symbol did not receive a value before its value was used, but it has not encountered any occurrence of its receiving a value yet.

Parameter-pattern is a pattern description which must match the next item following the occurrence of the symbol. It includes the number of parameters and the data-type of each one. This is used, if found, to make sure the appropriate number of parameters are passed to a procedure or an array reference.

The values associated with these attributes are set during the semantic phase of the parsing mechanism. During this phase certain checking is also done. Giving the user a warning about the possible use of a symbol before it has a value is an example of this. More will be said about this in the next chapter.

The PCM formalism is the template for generating structures in PCMREP. The PCM methods, discussed in the next chapter, use both the formalism and the structure as a basis for their actions.

CHAPTER 5

THE SYSTEM ACTIVITIES

The PCM formalism and the PCM structure (PCMREP) have been presented in order to describe the PCM activities based on them. These activities are presented below in four logical groups: functions which move the focus of attention within PCMREP; functions which modify PCMREP; functions which present, in different formats, program segments which reside in PCMREP; and functions to parse, check, and correct new program segments for placement in PCMREP. These activities are described functionally, since their invocation is terminal-dependent, i.e., at the user-interface level, the act of modifying a program segment would be quite different if the terminal is a graphics terminal with a light-pen and function box than if it were a teletype. If a function's activities are based on the PCM formalism, its description will be divided into five categories -- one for each of the PCM formalism's metaconstructs.

Before describing the PCM's activities, some notational considerations are necessary. A PCMREP node is represented as an italicized uppercase letter from the end of the alphabet (e.g., *X* and *Y*). If a function returns a value, it will be represented as

VALUE <- FUNCTION-NAME (PARAMETERS)

Otherwise it will be represented as just

FUNCTION-NAME (PARAMETERS)

Examples reflecting an action in the example session in Section 1.3 will be given.

5.1 *POSITIONAL FUNCTIONS*

These functions are used to move through PCMREP and change the focal point of the PCM activities. The name of the global definition being manipulated (i.e., the name of function or procedure currently being edited) is held in the Current Definition cell (CD) and a stack of pointers into CD's program structure is held in the CIP. The top of CIP is the pointer into the program structure which is of current focus. The following functions are used to manipulate CD and CIP, traverse the program structure, and search the program structure.

Functions:
SET-CD (symbol)
symbol <- FETCH-CD

These functions, respectively, set and retrieve the CD, i.e., the global entry on which the PCM activities are currently operating. In the example, whenever a DEFINE command was issued, the PCM did a SET-CD for the item defined.

Functions:
PUSH-CIP(*X*)
X <- POP-CIP
X <- TOP-CIP
boolean <- EMPTY-CIP?

The CIP is a stack of pointers internal to the CD, where the bottom of the stack is the top of the definition. These functions manipulate the CIP stack.

Function:
X <- LOCATE (string search-technique)

LOCATE finds the lexical unit specified by *string* in the structure pointed to by TOP-CIP by the search-technique specified (either DEPTH-FIRST or BREADTH-FIRST). It then returns its first nonsingular father (that is, the first father which has other sons). Usually, the first father of the found string is the node TERMINAL; therefore LOCATE returns the node which also contains the found string's siblings. In the example session

```
FIND BEGIN
```

caused the following to be executed:

```
PUSH-CIP ( LOCATE ( "BEGIN" DEPTH-FIRST ) )
```

followed by a short-form printing of the new TOP-CIP.

Functions:
X <- FATHER (*Y*)
X <- FIRST-SON (*Y*)
X <- LAST-SON (*Y*)
X <- LEFT-BROTHER (*Y*)
X <- RIGHT-BROTHER (*Y*)

These functions provide general relational movements through PCMREP. In the example session

```
GO 2
```

caused the following to be executed:

```
PUSH-CIP ( RIGHT-BROTHER ( FIRST-SON ( TOP-CIP ) ) )
```

List of Positional Functions

SET-CD (symbol)
symbol <- FETCH-CD
PUSH-CIP(*X*)
X <- POP-CIP

```
X <- TOP-CIP
X <- LOCATE ( string search-technique )
X <- FATHER ( Y )
X <- FIRST-SON ( Y )
X <- LAST-SON ( Y )
X <- LEFT-BROTHER ( Y )
X <- RIGHT-BROTHER ( Y )
```

5.2 MODIFICATION FUNCTIONS

The above functions were dependent only on PCMREP, but the modification functions base their operation not only on PCMREP but also on the PCM formalism. Each function operates differently depending on which metaconstruct was used for the parse into PCMREP. Therefore, there are five parts to the description of each function; each part describes the action of the operation for one metaconstruct. Also, the modification functions always preserve the PCMREP structure as if the program represented had been parsed into PCMREP all at once.

Function:
DELETE (*X...Y*)

DELETE causes the nodes specified (they are the sons of TOP-CIP and any number may be specified) and the associated semantic information to be removed from PCMREP. But it must first make sure the deletion will still leave a syntactically correct PCMREP of the particular programming language. This is accomplished based on the metaconstructs. We will discuss these on a case-by-case basis.

DELETE in an Ordered Sequence:

Deletion can only be accomplished if it is deleting an element of the Ordered Sequence which is an Alternating Sequence with an EMPTY Alternative. For example, if the rule was

```
if-statement -> [ 'IF' boolean-expression 'THEN' statement
      { [ 'ELSE' statement] | EMPTY } ]
```

then the user could delete the ELSE and its associated statement (i.e., DELETE (5 6)). Although this is a simple rule, a severe change in PCMREP may be necessary. It should be restated here that the philosophy of the PCM modification functions is that the PCMREP will always reflect a structure as if the programmer had had his whole program parsed at one time. Therefore, if the ELSE clause being deleted is a member of an IF statement which is the THEN clause of another IF statement, then the following restructuring takes place (using square brackets as metaconstructs to show association):

[IF A THEN [IF B THEN C ELSE D] ELSE E]

becomes

[IF A THEN [IF B THEN C ELSE E]]

That is, if

FATHER (TOP-CIP)

was produced by the same metaconstruct rule (e.g., if-statement) as seen in Figure 6, then the deletion of ELSE.2 and statement.4 ● must cause a restructuring to match what PCMREP would be if the user had input if-statement.1 directly. Figure 7 (using the same subscripts as Figure 6), shows that PCMREP. The ELSE.1 and statement.2 were "pushed down" into if-statement.2 DELETE will detect this, perform the restructuring, and inform the user of it.

DELETE in an Alternating Sequence:

To maintain a legitimate PCMREP, the deletion must specify a pair of adjacent nodes, that is, a node for f.1 and one for f.2. These can be specified in either order. For example, a legitimate PCMREP for the production rule

expr -> < identifier { '+' | '-' | '*' | '/' } >

can be seen in Figure 8. The PCM would accept as legitimate either DELETE (1 2) or DELETE (2 3), but not DELETE (3). If there is only one item (i.e., an instance of f.1), then the deletion is legitimate only if the Alternating Sequence is one of the forms in an Alternative Set which contains an EMPTY alternative.

DELETE in a Bracketed Sequence:

A Bracketed Sequence is made up of a form surrounded by start and finish markers (the brackets) such as

[) 'BEGIN' < statement ';' > 'END' (]

The inner form (i.e., the second element of the Bracketed Sequence -- in this case it is < statement ';' >) is treated as a part of the Bracketed Sequence, as was seen in the example session when the PCM printed

begin <<STACKINDEX ... 1>> ; <<STACK ... ITEM>> end

For deleting any element of the inner form, the rule for that metaconstruct applies. For example, DELETE (2 3) would be a legitimate operation removing the first statement and the semicolon of the Alternating Sequence. If the inner form was an Ordered Sequence, then the "DELETE in an Ordered Sequence" rule applies. The deletion of the "brackets" is usually not allowed. For example, removing the parentheses surrounding a parameter list is not allowed. There are two exceptions.

● The subscripting notation is used in the figures for clarity.

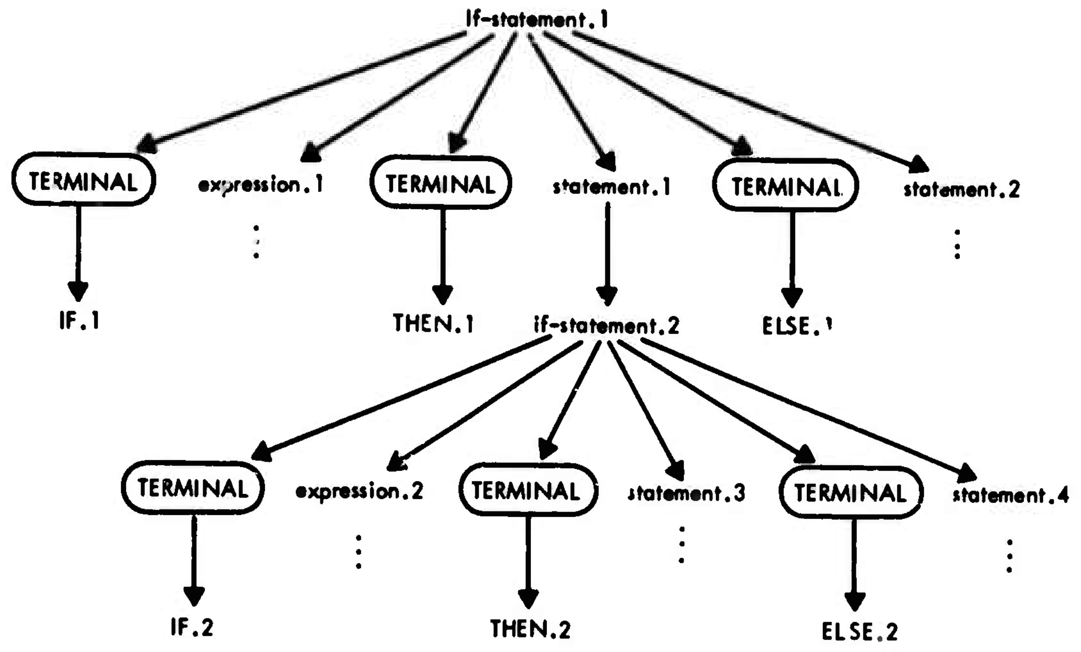

Figure 6. Embedded ordered sequences of the same type.

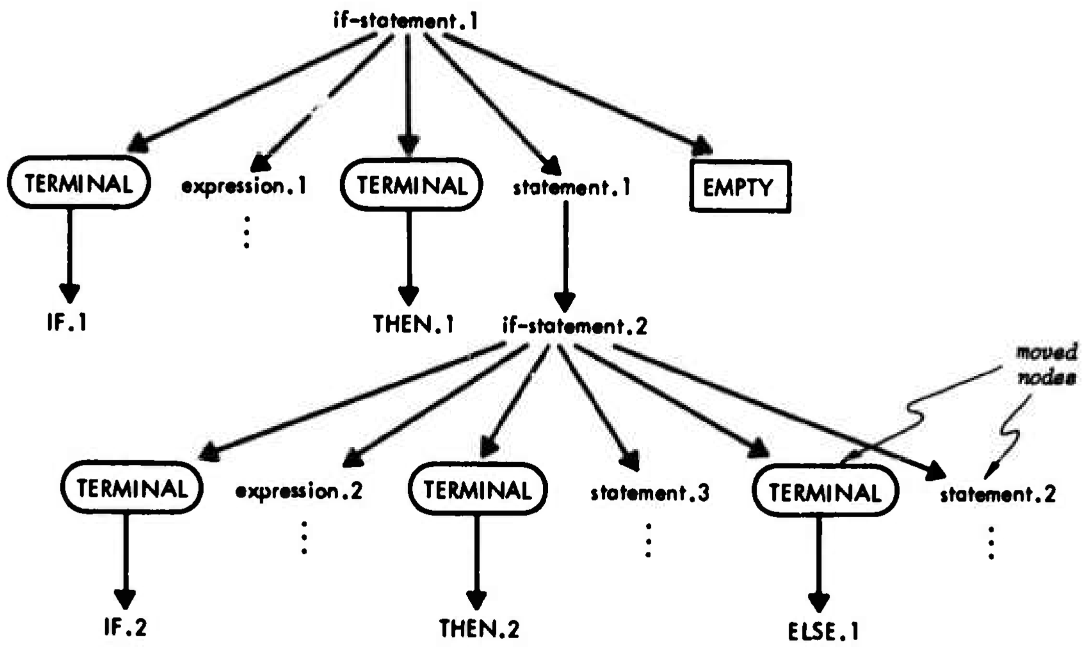

Figure 7. Embedded ordered sequences of the same type with an EMPTY alternative.

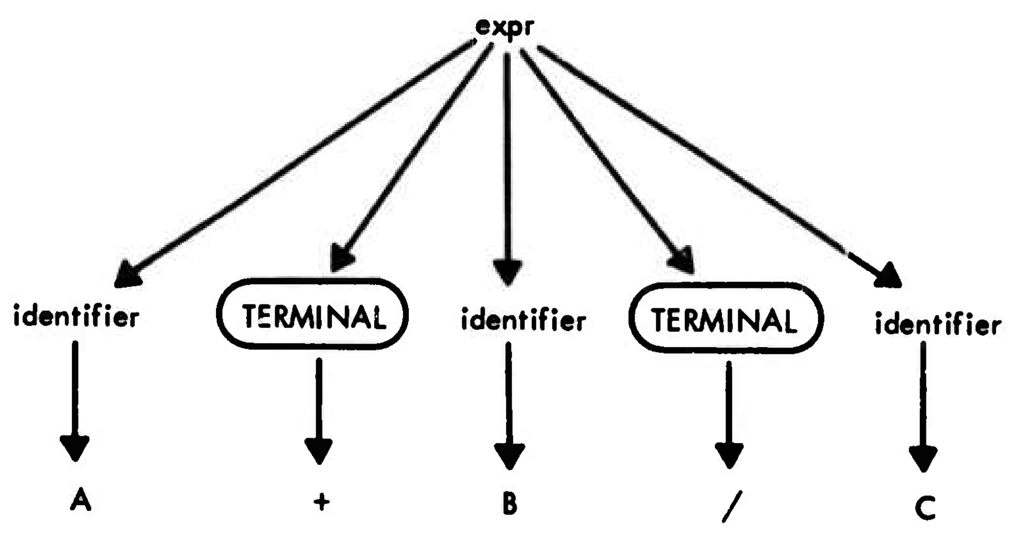

Figure 8. A legitimate PCMREP for expr -> < identifier {'+' | '-' | '*' | '/'} >

First, in cases such as arithmetic expressions, where a Bracketed Sequence is used to describe subexpressions, it should be possible to delete parentheses. This is accomplished by the following check of PCMREP. If the FATHER of the Bracketed Sequence is the same as the inner form, then they can be removed and the inner form is raised, removing the inner form. That is, if this was a production rule

```
arith-expression ->
     < { identifier | [) '(' arith-expression ')' (] }
          { '+' | '-' | '*' | '/' | '↑' } >
```

then a possible PCMREP can be seen in Figure 9. The user could then legitimately cause a DELETE (3 5) to be issued which would cause the structure to change as seen in Figure 10.

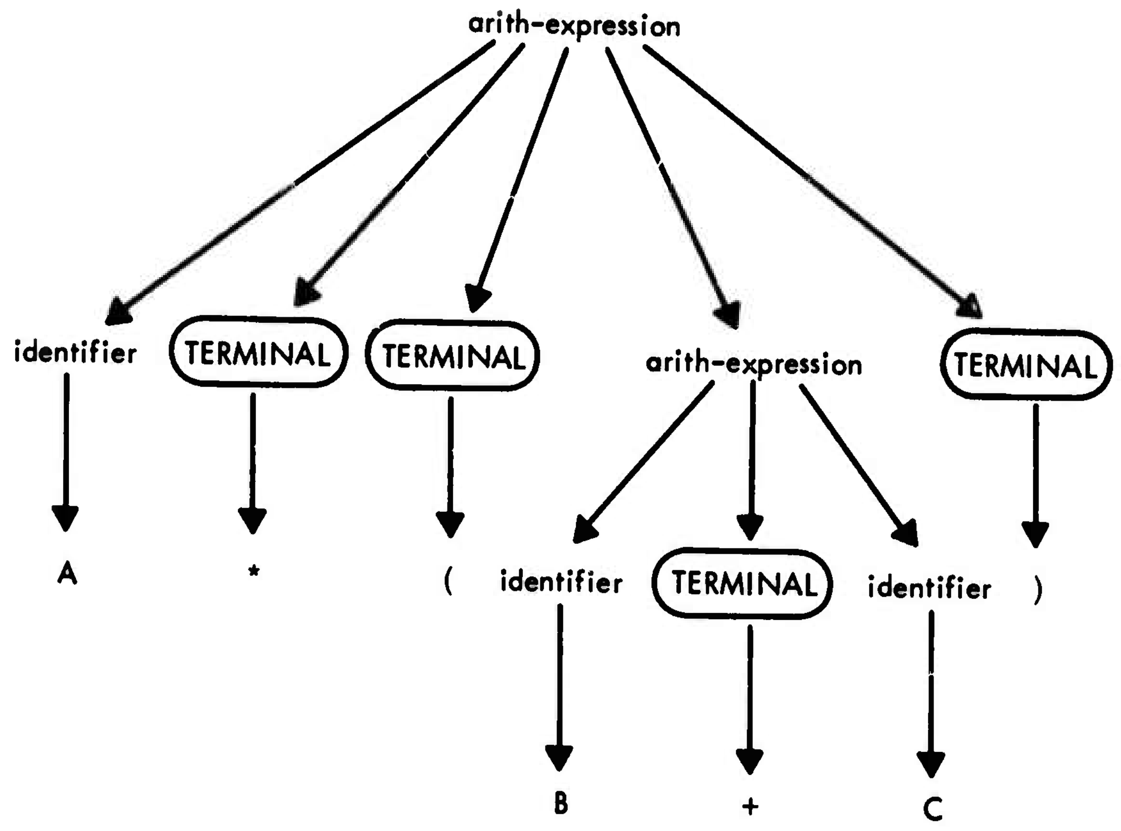

Figure 9. A legitimate PCMREP for

arith-expression -> < {identifier | [) '(' arith-expression ')' (]}
{'+' | '-' | '*' | '/' | '↑'} >

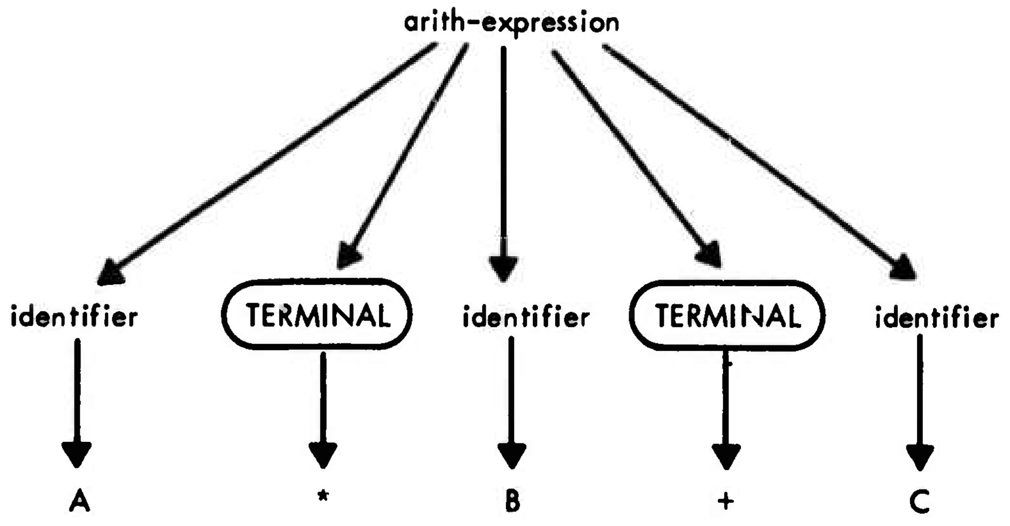

Figure 10. Modified PCMREP from figure 9.

The second exception deals with cases such as BEGIN blocks which are the statement following the ELSE in an IF statement. Then deleting the BEGIN-END pair should be allowed. This is accomplished by checking several criteria. First, the inner form must be an Alternating Sequence whose first form is the same type as the Bracketed Sequence. Second, the grandfather must be the same Bracketed Sequence. This can be seen in Figure 11 where statement.1 and statement.4 are both begin blocks. If these criteria are met, then the deletion is legitimate and the following operations take place. Replace the Bracketed Sequence with the first set of Alternating Sequence (i.e., the first f.1 and f.2). Insert the remainder of the Alternating Sequence after the grandfather. This can be seen in Figure 12.

DELETE in a Repeating Sequence:

If the form to be deleted is another metaconstruct, then the deletion rule for that metaconstruct is applied. If not, then any deletion is legitimate unless it is deleting the last entry. In this case, the Repeating Sequence must be an item in an Alternative Set with an EMPTY alternative.

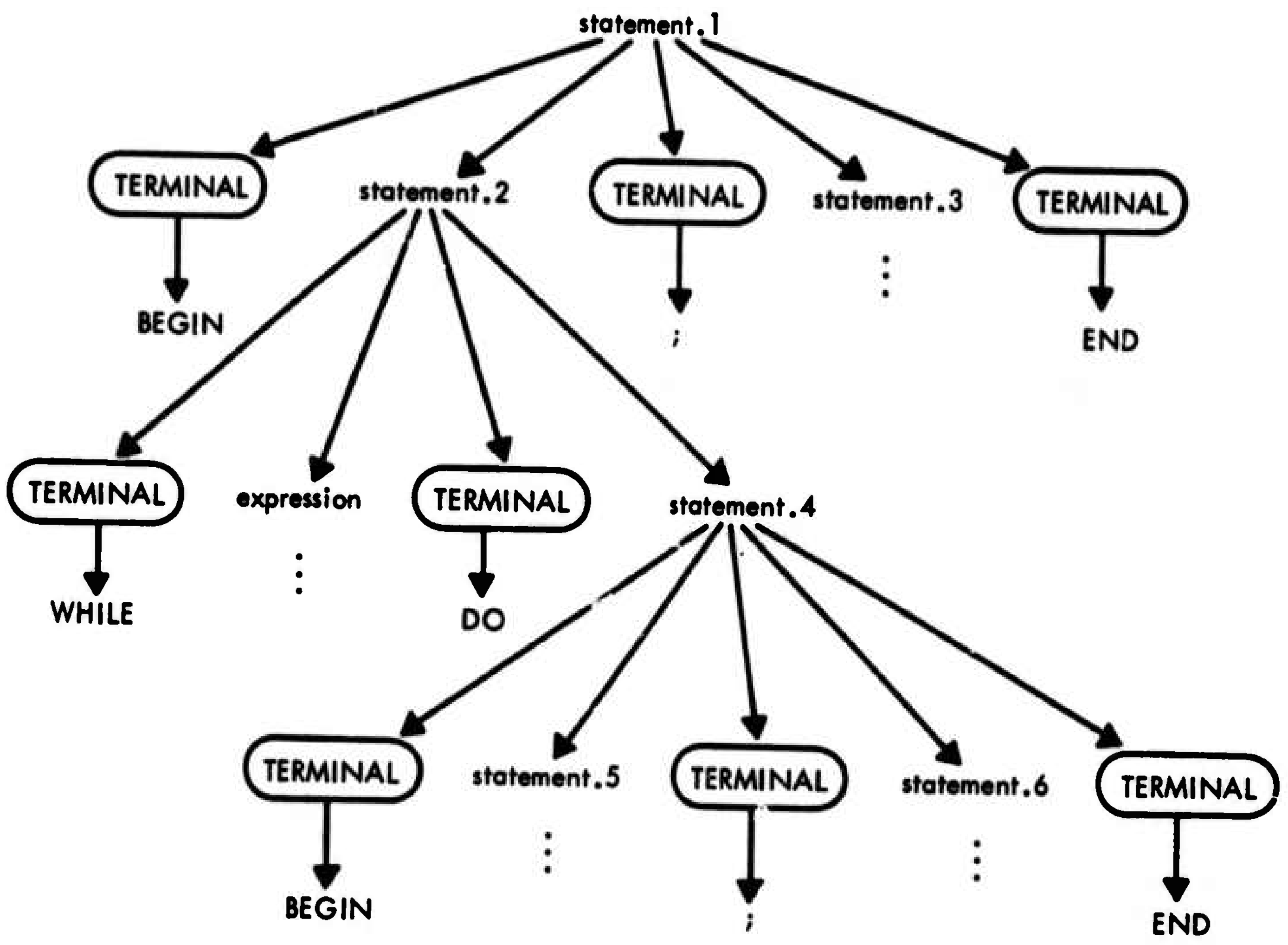
statement.1
TERMINAL
BEGIN
statement.2
TERMINAL
;
statement.3
TERMINAL
END
TERMINAL
WHILE
expression
TERMINAL
DO
statement.4
TERMINAL
BEGIN
statement.5
TERMINAL
;
statement.6
TERMINAL
END

Figure 11.

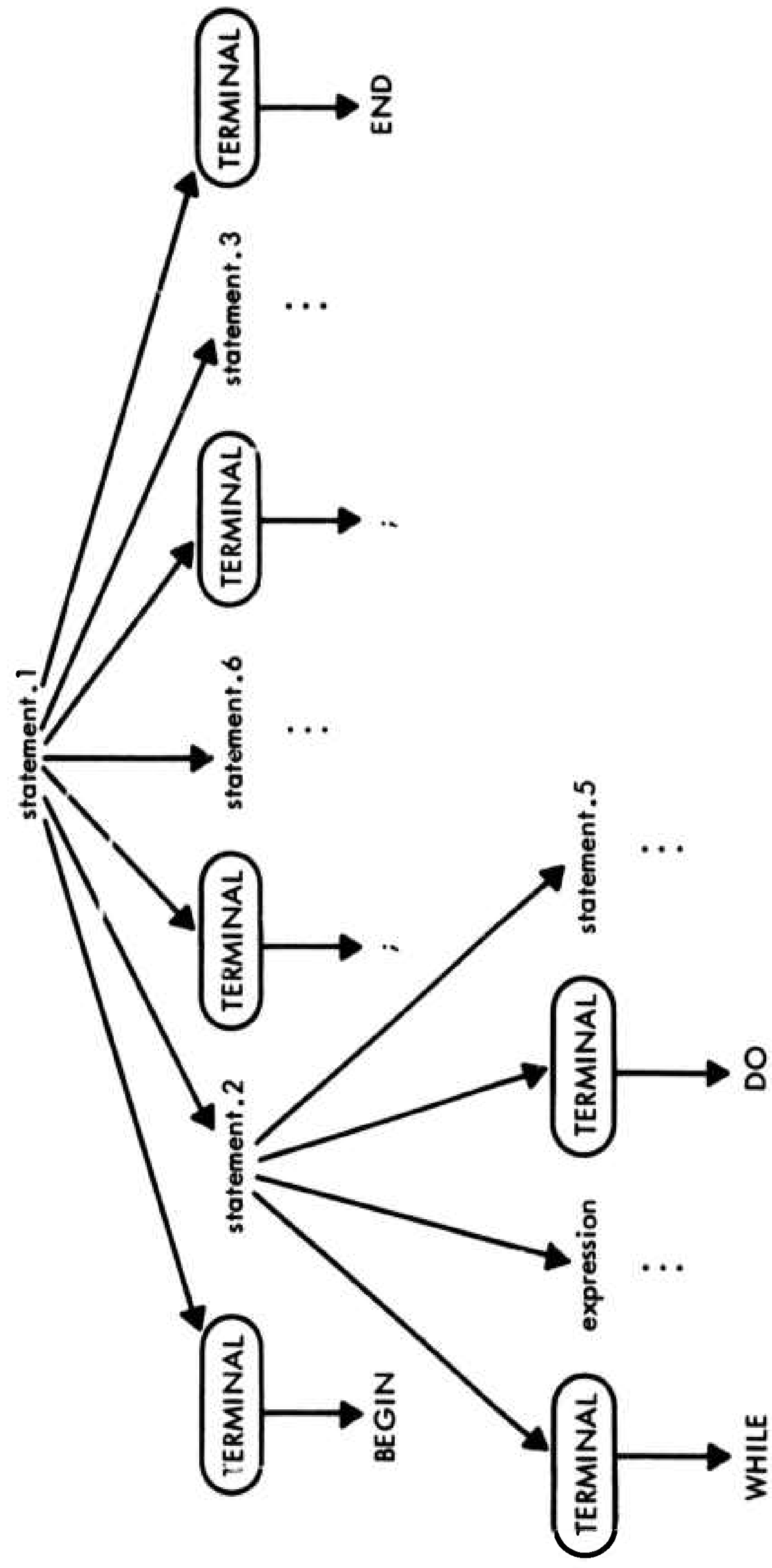

Figure 12. Modified PCMREP from figure 11.

Function:
INSERT-BEFORE (string *X*)

The principal function of INSERT-BEFORE, like several other functions which follow, is to build the appropriate production rule to give to the parsing mechanism. That is, it must decide what are the valid structures that can be inserted before the specified node. As with DELETE, the function of INSERT-BEFORE is dependent on the current parse node (CIP).

INSERT-BEFORE in an Ordered Sequence:

Since, with one exception, an Ordered Sequence is a *required* list of forms, insertion is, in general, not legal. However, this exception gives the PCM one case to consider. That case is when the node before the specified node is EMPTY. This means the node was created from an Alternative Set with an EMPTY alternative. An example of this can be found in programming languages which can have a label before a statement. Then the rule for statement may be the following:

```
statement -> [ { [label ':' ] | EMPTY }
      { assign-statement | if-statement | ... } ]
```

If the Alternative Set is of length two, then the first form is given to the parsing mechanism, e.g., from the above example, the parser would be given

```
goal -> [ label ':' ]
```

Otherwise, the Alternative Set is modified by removing the EMPTY alternative and then given to the parser.

INSERT-BEFORE in an Alternative Set:

This is never valid, since there is only one form which was selected. The user is probably in the wrong location in PCMREP, so the input string is saved and the user is asked if he would like to change the CIP and redo the command. The user can abort or change the CIP.

INSERT-BEFORE in an Alternating Sequence:

An Alternating Sequence (< f.1 f.2 >) is a sequence of forms alternating between f.1 and f.2 and ending with an f.1. The PCM user can insert before either an f.1 or f.2. The function builds two different rules based on whether it is before an f.1 or f.2 to maintain the correct alternations. If the insertion is before an f.1, then the parsing mechanism is started on

```
goal -> [* [ f.1 f.2 ] *]
```

If it is before an f.2●, then the production is

● The goal is one or more occurrences of the Ordered Sequence f.1 and f.2.

goal -> [* [f.2 f.1] *]

This causes the appropriate sequencing for the insertion.

INSERT-BEFORE in a Bracketed Sequence:

In a Bracketed Sequence ([) t.1 f.1 t.2 (]), it is not valid to insert before t.1. If the insertion is before t.2, then this is converted into INSERT-AFTER of the inner form (f.1). The appropriate action is taken based on the structure of f.1. This is also true for an insertion before f.1.

INSERT-BEFORE in a Repeating Sequence:

This is the easiest to handle, since a Repeating Sequence is only a repetition of one form. The same rule (i.e., [* f.1 *]) is given to the parsing mechanism.

Function:
INSERT-AFTER (string *X*)

INSERT-AFTER is similar to INSERT-BEFORE. In most cases its functions are identical to INSERT-BEFORE except that it performs its validation checks for what can come after node ***X*** rather than before it.

Function:
REPLACE (string *X*)

REPLACE is relatively simple, since whatever new input is to replace the old node must be of the same type (i.e., the same nonterminal) as that node. The parser is given whatever rule or metaconstruct caused the initial node to be generated. Notice that the node must be either a production rule or a metaconstruct. Terminals with no alternatives are not allowed to be replaced. For example, if the rule was

```
if-statement -> [ 'IF' boolean-expression 'THEN' statement
     { [ 'ELSE' statement ] | EMPTY } ]
```

then the PCM would accept

REPLACE ("A < B" 2)

but not a command dealing with node 1.

Function:
EMBED (string *X*)

EMBED is similar to REPLACE in that the new partial PCMREP must be of the same type as the old node. The parsing mechanism is started with the same nonterminal after replacing the specified character with a flag the parser is aware of and a pointer to that node in PCMREP.

When the parser reads the flag it checks to see if the nonterminal in PCMREP is the same as the next item to be parsed, the parser skips that action and adds that node to the partial PCMREP it is building.

Function:
EXTRACT (*X*)

EXTRACT is the opposite function of EMBED. It is designed to remove structure. Node *X* should be of the same type nonterminal as the one pointed to by the CIP. If so, the function deletes from PCMREP all the other nodes and makes node *X* the new CIP. Notice that recursion in the formalism is implied in both EMBED and EXTRACT.

Function:
DEFINE (string)

This function is used to define a new global symbol and its value (usually a program segment). It starts the parser on the initial nonterminal, then sets the CD and CIP pointers to the new definition.

List of Modification Functions

DELETE (*X*...*Y*)
INSERT-BEFORE (string *X*)
INSERT-AFTER (string *X*)
REPLACE (string *X*)
EMBED (string *X*)
EXTRACT (*X*)
DEFINE (string)

5.3 *PRESENTATION FUNCTIONS*

The presentation functions are provided to give the PCM user information about his current state of program development. This information is either program information (unparsing) or status information such as undefined symbols.

Status information is based on two functions which operate on the access nodes where the information is stored. They are

Functions:
list-of-symbols <- FETCH-SYMBOLS (access-node)
value <- RETRIEVE (access-node symbol attribute)

FETCH-SYMBOLS returns a list which contains each symbol in the access node specified. RETRIEVE returns the value of the attribute specified for that particular symbol. Several examples of their use by the PCM could be seen in the example session. For example, UNDEFINED? has the following definitions:

```
FOREACH SYMBOL IN FETCH-SYMBOLS ( GLOBAL ) DO
```

```
IF RETRIEVE ( GLOBAL SYMBOL DECLARED ) = NO
     THEN PRINT (
          RETRIEVE ( GLOBAL SYMBOL DATA-TYPE )
          SYMBOL ) )
```

This example should be sufficient for the reader to extrapolate their usage in other status functions.

As was seen in the example session in Chapter 1, there are two forms of program presentation: long and short. The long form presents the entire program pointed to by the CIP in a structured form (i.e., not a linear string of lexemes). The short form also presents the program pointed to by the CIP but in an abbreviated form showing the structure but not the entire program. Both unparse PCMREP.

Function:
UNPARSE-LONG (*X*)

This function processes the specified part of PCMREP outputting, in structured form, the lexical units. The methods for outputting them in structured form is again based on the metaconstructs of the formalism. These methods recursively call UNPARSE-LONG on some of the metaconstruct's elements.

Before describing these methods, some functions used by UNPARSE-LONG and some global variables will be discussed. UNPARSE-LONG needs to know the line length (LL), the current left margin (CLM), the current indentation (i.e., where text starts on a line) (CI), the remain text area on a line (REM), and the number of spaces for a tab stop (TAB). NEWLINE, an affiliated function, prints a carriage return and sets CLM to the value of CI. OUTPUT prints the lexical unit, increments CLM and decrements REM.

If this is the first time this structure has ever been unparsed in long form, then a pass through it must be made leaving information at each nonterminal. Starting at the leaf nodes the information needed is the total number of characters for all lexical units below it, the number of lexemes, and a flag associated with this information is set to VALID. After this is done, the function LENGTH (*X*), using this information, is used by UNPARSE-LONG to give the length in characters of the structure if printed on one line (i.e., total characters + (lexical units - 1)).

After the modification is made, each modification function (see Section 5.2) ascends PCMREP, changing the associated flags to NOTVALID. The next time UNPARSE-LONG is initiated, it checks this flag and descends PCMREP, only changing the parts that are marked NOTVALID by correcting the information and changing the flag.

The basic approach of UNPARSE-LONG's operation on node *X* is

1. If *X* is a leaf node, then PRINT (*X*) and return.

2. If LENGTH (*X*) is less than REM, then PRINT all the lexical units in *X* separated by spaces and return.

3. Apply the appropriate method described below to *X* based on which metaconstruct is associated with *X* and then return.

UNPARSE-LONG on an Alternating Sequence:

Using the notation < f.1 f.2 >, the approach is

1. If f.2 is a language constant, then for each f.1 f.2 pair UNPARSE-LONG (f.1), PRINT (f.2), and then NEWLINE. Finally return. This is for the cases such as < statement ';' >.

2. Otherwise, for each f.1 and f.2 see if LENGTH(f) is less than REM. If it is, then UNPARSE-LONG(f). If not, increase CI by the value of TAB then NEWLINE and UNPARSE-LONG(f). Finally reset CI and return.

UNPARSE-LONG on a Bracketed Sequence:

Using the notation [) T.1 F.1 T.2 (] the approach is

1. If t.1 is an alphabetic language constant, then PRINT (t.1), increase CI by TAB, NEWLINE, UNPARSE-LONG(f.1), decrease CI by TAB, NEWLINE, PRINT(t.2), and return. This is for cases such as

   [) 'BEGIN' <statement ';' > 'END' (]

   and would cause the following output:

```
BEGIN
    statement.1 ;
    statement.2 ;
            .
            .
            .
    statement.n
END
```

2. Otherwise PRINT(t.1), PRINT(space), increase CI by LENGTH(t.1)+1, UNPARSE-LONG(f.2), reset CI, PRINT(t.2), NEWLINE, and finally return. This would handle such cases as

   [) '(' < parameter ',' > ')' (]

   and would cause the following output:

```
FOO ( parameter.1 ,
      parameter.2 ,
            .
            .
            .
      parameter.n )
```

UNPARSE-LONG on a Repeating Sequence:

A Repeating Sequence is just a indefinite sequence of forms, therefore for each form UNPARSE-LONG(f) and NEWLINE. Finally return.

UNPARSE-LONG on an Ordered Sequence:

An Ordered Sequence uses the same approach as part 2 of the approach for Alternating Sequences.

These methods produce a structured output which reflects the structure of the programming language described in the formalism.

Function:
UNPARSE-SHORT (*X*)

This short form of program presentation is used to show the PCM user the structure of his program quickly and concisely. The approach is to output each son of *X* in order. If the son is a single lexical unit, then output it. For larger sons (i.e., more than one lexical unit), the first and last lexical units are printed with the expert-supplied start, middle, and finish characters. An example was seen in the example session when the PCM output

begin << STACKINDEX...1 >> ; << STACK...ITEM >> end

Using different approaches, both program presentation functions reflect to the PCM user the structure of his program.

List of Presentation Functions

list-of-symbols <- FETCH-SYMBOLS (access-node)
value <- RETRIEVE (access-node symbol attribute)
UNPARSE-LONG (*X*)
UNPARSE-SHORT (*X*)

5.4 THE PARSING MECHANISM

The parsing mechanism is a two-stage process which maps a character string into PCMREP. The first stage is what is normally thought of as the parse stage. The second stage is the semantic evaluation and checking phase. Only after both stages have been executed is the PCM satisfied in the validity of the new PCMREP. Each stage will be discussed separately.

The Parsing Stage

Function:
partial-PCMREP <- PARSE (string goal)

Syntax-directed parsing techniques are so widely found in the literature that it is not necessary to discuss them here.● Syntax-directed error correctors are also found in Computer Science literature (see Section 2.3) but the reader may not be as familiar with these, so a short description of recent correctors (over the last four years) will be given, since they have similar approaches.

The corrector is initiated when the parser can no longer continue. Notice that this may not be the point where the error is but only where an error is detected. Consider the following program segment for a ALGOL-like language:

```
... ; EOF THEN ...
```

When the parser cannot continue at the THEN, the corrector is invoked. Presumably the "best" correction is to insert an IF after the semicolon, which implies the parser must back up. Or consider a worse case.

```
... ; IF EOF THEN  X := Y; Y := Z  END ELSE ...
```

If the END erroneously closed off some previous BEGIN, the error is encountered at the ELSE. There is a "best" correction, where "best" here is defined as the (1) minimum change to the original input and (2) insertion is "better" then deletion; insert a BEGIN after the THEN. Each corrector has developed different heuristics to determine how far to back up before correction and heuristics to then try to make the correction which includes spelling correction of reserved words.

Unfortunately, they all suffer from the same phenomenon -- since they are not guaranteeing a "correct" correction, they may have done damage rather than rectification, causing an ***avalanche*** of errors for further input. Here PCM excels. The PCM user is always asked to confirm any proposed change to his input. If he rejects the proposed change, he can edit his input and restart the process. Also, the PCM will not allow the corrector to spend an exorbitant amount of time looking for a correction. If this occurs, the PCM user is asked to either edit the input (with the current parse state at the detected error point given him) or abort the parse.

When the parse is complete, the result is an incomplete PCMREP without access nodes. These are added to the PCMREP during the semantic checking stage. But first this partial PCMREP must be added to the more global PCMREP. This is the responsibility of the modification function which initiated the parse. Each modification function knows how to do this addition to PCMREP. For example, when INSERT-BEFORE in an Alternating Sequence gave the parser

```
goal -> [* [ f.1 f.2 ] *]
```

obviously a Repeating Sequence is not the desired end result. INSERT-BEFORE removes the nonterminal, goal, and its associated metaconstruct. It then links up the appropriate right and left brothers of the Alternating Sequence. Once the insertion is accomplished, the semantic checker takes over.

● Suffice it to say that a bottom-up SLR(1) syntax-directed parser was developed which used the PCM formalism.

The Semantic Phase

Function:
SEMANTICS (*X*)

As was stated in Chapter 3, the expert also supplies some semantic information in his language description. This information is enclosed in parentheses and can follow any metaconstruct. It is then associated with that metaconstruct. The semantic checker is started at the new node in PCMREP and, processing top-down, scans the new PCMREP segment. Upon encountering one of these pieces of semantic information, the semantic checker takes appropriate action. These actions will be discussed as each type is presented.

Semantic Attribute:
(NEW ACCESS NODE)

This causes a new access node● to be added to PCMREP above the metaconstruct and linked up to its parent access node. An ALGOL language description would contain the following:

[) 'BEGIN' < statement ';' > 'END' (] (NEW ACCESS NODE)

This would cause a new empty access node to be inserted just above the block in PCMREP. Now symbols referenced below will use this access node.

New symbols are inserted into this access node by

Semantic Attribute:
(DECLARE symbol attribute:value ... attribute:value)

Whenever this is encountered, the new symbol is inserted into the first parent access node encountered and the attributes for that symbol are added. For practical purposes there must exist a binding mechanism to retrieve information from PCMREP. This is accomplished by a FOREACH function whose first parameter is the nonterminal which represents a lexical unit. Each lexical unit is taken, one at a time, and replaces the nonterminal in the semantic statement, which was the second parameter to FOREACH. For example, a SAIL declaration would be

```
        decl-stmt -> [ type < identifier ',' > ]
                ( FOREACH identifier ( DECLARE identifier data-type:type ) )
```

DECLARE also updates several attributes automatically (i.e., ***declared***, ***declaration-pointer***, and ***context-mode***). Also notice that type was automatically bound to its lexical value.

Semantic Attribute:
(SET symbol)

This causes ***set-flag-dependency-set*** to be updated to contain this node. This later is used in determining the possibility of a symbol being used before it has a value. It may appear in the formalism as

● It is assumed that the reader is familiar with access nodes described in Chapter 4.

```
assign-statement -> [ identifier ':=' expression ]
     ( SET identifier )
```

or possibly

```
formal-parameter-list -> { [ type < identifier ',' > ] |
     [ 'REFERENCE' type  <identifier ',' > ]
          ( FOREACH identifier ( SET identifier ) ) }
```

Semantic Attribute:
(USED symbol)

This causes the *occurrence-dependency-set* to be updated and a check made to see if it has been set. If not, a warning is given to the user to that effect.

It is possible that, for either SET or USED, the symbol is not in the current access node. The semantic checker considers this an error and invokes the semantic corrector. The semantic corrector first searches up PCMREP looking for an access node containing the symbol. If it finds one, the access nodes are linked and the corrector returns. If it does not find the symbol, it then tries to correct a presumed spelling or typing error by seeing if it is a close match® to any symbols in its access node path. If it finds one it asks the user to confirm this correction. If no close match was found, it asks the user if the symbol is global. At this point the user can either confirm that it is or fall into a lower edit, create the symbol, and upon returning the corrector confirms that there is indeed a new symbol in the access node path.

Semantic Attribute:
(PARAMETER symbol data-type)

This causes *occurrence-pattern* to change by appending the data-type specified. A FOREACH is used to iterate through the formal parameter list. Whenever that symbol is then used in a SET or USED semantic statement, a check is made on the number of parameters which follow it in PCMREP. Of course, for languages which allow a variable number of parameters on function calls, this feature cannot be used.

This ends the discussion of the semantic checker. However, since structure may be removed from PCMREP, so must the semantic information be removed. This is accomplished by

Function:
UNDO-SEMANTICS (*X*)

which removes all semantics added to PCMREP for the node and all of its descendants. It traverses the tree in the opposite order of the semantic checker and removes all the information inserted into the PCMREP by the semantic checker. There are a few cases which might cause possible errors. These cases are caught by UNDO-SEMANTICS and are as follows:

® Closeness is measured by the number of disagreements between the two words being compared divided by the length of the longest word. This gives a percentage of "agreement". Currently, if the agreement is greater than 70 per cent, the words match.

- If the *set-flag-dependency-flag* becomes empty, a check is made to see if the *occurrence-dependency-set* is also empty. If not, then a warning is made to the programmer about the symbol's possible use before it received a value.

- If the *declaration-pointer* is deleted and either the set or used sets are not empty, then a search is made up the access nodes for the same symbol in a higher context. If found, the access node information is changed to link up to the higher symbol. Otherwise, the error is presented to the user for his rectification.

List of Parsing Functions

partial-PCMREP <- PARSE (string goal)
SEMANTICS (*X*)
UNDO-SEMANTICS (*X*)

5.5 THE IMPLEMENTATION

A prototype version of PCM has been implemented in INTERLISP● on a PDP-10 TENEX. PCMREP was implemented as a n-ary doubly-linked list structure for ease of backward and upward movement. The symbol tables within the access nodes had the form of LISP's property lists (although they were not associated with any LISP atoms, as real property lists are). The attributes for the symbols were similarly implemented, and both were accessed by the functions PUTL and GET.

The parser was implemented by writing five functions which correspond to the five metaconstructs of the PCM formalism, a function to scan the input and return lexemes●●, and a predicate function to confirm that the next lexeme in the input string was a particular language constant, such as semicolon. RETFROMs were used to back up the parsing process.

The other functions described in this chapter were implemented as described, with the exception that some of the simple straightforward algorithms for some cases, not needed for the example session in Chapter 1, were left out of this prototype implementation.

The definition of PASCAL was hand-translated from the formalism to calls on the functions which reflected the metaconstructs, and the language constants (enclosed in quotes in the formalism) were converted to calls on the predicate function that would return T if the constant matched the next lexeme in the input string.

The system ran interpretively and obviously was not an efficient implementation. But it was sufficient to show the feasibility of such a system for helpful aid in the creation and modification of syntactically correct program.

● In this discussion, it is assumed the reader is familiar with LISP.

●● The scanner recognized identifiers, reserved words, numbers, operators, and constants by reading LISP atoms through a special readtable.

THE SYSTEM ACTIVITIES

The PCM formalism, PCMREP, and the PCM activities provide the facilities to achieve the structured creation and modification of programs in many programming languages. Chapter 6 will summarize this along with the reasons for a LIBNI approach to the problem. Chapter 6 also presents the author's conclusions and some suggestions for future work.

CHAPTER 6

SUMMARY AND CONCLUSIONS

6.1 ENVIRONMENTS

Several well-known computer scientists have argued along with the author (see Section 1.1) that the next big advancement in the area of computer software production is monolingual programming environments with sophisticated facilities for program development. These facilities should contain and exhibit knowledge about their tasks which was previously maintained in the programmer's head. This allows the user to concentrate more on the programming problem rather than the programming effort and should allow an increase in the complexity of the problem domains. This increase is the main reason for programming environments with sophisticated facilities. Although it is difficult to generalize at this level of discussion, an advanced facility guides the user in his performance of the task while, without interfering with his performance, it prevents him from making errors.

Obviously, this is an immense task to perform for all programming languages. The author's solution to this dilemma is to make as many facilities as possible language-independent. To accomplish this, the "knowledgeable" tasks performed by a facility must be conceptualized in a general way into the realm of programming languages rather than one particular language. Then by some form of language description appropriate to the task, the system should tailor its conceptualization to a particular programming language. This approach the author has labelled LIBNI; it stands for Language-Independent But Not Ignorant.

Programming environments should be ever expanding their facilities as new techniques are designed such as program verification (which possibly will be a viable tool in a few years). This expansion is exhibited in INTERLISP, which is a very sophisticated programming environment for the programming language LISP and was the initial inspiration for the author's research. Unfortunately, it is not LIBNI-based. For a LIBNI programming environment ever to appear, LIBNI facilities must first be designed and shown to be possible. This was the impetus for this report.

6.2 PROGRAM CONSTRUCTION AND MODIFICATION: A SUMMARY

After extensive investigation in LIBNI programming environments as a whole, the author selected the specialized area of program construction and modification, for several reasons.

- This area is the most neglected and historically unrecognized as a problem area in computer science.

SUMMARY AND CONCLUSIONS

- The facility is capable of being used without the need of a programming environment. That is, it could replace the normal text editor now being used to construct programs in the average interactive "general-purpose" computing environment and greatly improve the phase of program construction and modification.

- It is the one component of a programming environment that, with current knowledge, could be designed in a language-independent but not ignorant manner. That is, the knowledge needed for program construction and modification can be abstracted.

- During the time the author spent programming extensively while a professional programmer, he found the errors encountered during later phases which could have been prevented by a good facility for this phase of program development to be the most irritating of his programming activity.

The Goal

The goal was to carefully design an integrated LIBNI facility for program "editing". It should facilitate program construction by maintaining correct syntax and certain semantic consistencies (i.e., error prevention) while still allowing the user "free-form" program input. It should be able to present program structure in several forms; it should also be able to provide information concerning the use of symbols such as where and when they are defined and used. And it should react "intelligently" by giving warnings about possible inconsistencies in the use of symbols and by trying to correct simple input errors.

Attaining the Goal

To accomplish this in a LIBNI manner, the facility needs a description of a particular programming language's syntax and some of its semantics related to symbols. It needs an internal representation of this information for particular programs. And it must tailor its operations based on this language description. That is, it needs three items: a language description, a representation for holding programs, and "intelligent" and adaptable operations for program manipulation.

Since these pieces are interrelated, it was necessary to design the three items together, implement a prototype to test specific parts, review the results, and then revise the design when necessary.

The Resultant Design

An overview of the formalism for describing programming languages, the internal representation of programs in these languages, and the functions for manipulating and presenting these programs follows along with a system overview.

The author rejected BNF because its limited metaconstructs require the definition of an excessive number of nonterminals and distort some language constructs such as parameter lists.* A new formalism was designed similar to BNF but with an expanded set of metaconstructs which more directly reflect the syntax of high-level programming languages as it is conceptualized and manipulated by the programmer. There are five metaconstructs, which informally are

1. An ordered sequence of mandatory forms.

2. A set of alternative forms with a special form to indicate that it is not necessary to match any of the alternatives.

3. An ordered sequence of three forms where the first and last forms are language constants. This is useful for describing such items as block structure and parameter lists.

4. An indefinite number of alternations of two forms with the first form specified as being both the first and last item.

5. An indefinite number of repetitions of a single form.

The PCM** bases all of its activities on these metaconstructs, from parsing to the decision of whether a deletion of some part of a program will still maintain a legitimate program syntax.

A data base for holding the internal representation of programs (PCMREP) was also designed. It is a highly augmented n-ary tree with special nodes which contain extensive information concerning symbols which occur below it. They appear in the structure whenever a new naming context is present such as procedures or new blocks.

Methods were then designed for constructing and modifying programs for languages described in the formalism. These methods fall into four categories and are presented in Chapter 5 as sets of basic functions. A brief overview of the four categories follows.

1. PCMREP is designed to hold many program segments (e.g., procedures, functions, and global data definitions). A set of functions is used to focus attention on any of these that the programmer finds of interest and then within that definition maneuver to more local points.

2. A set of modification functions (deleting, inserting, replacing, embedding, extracting, and defining) was designed whose operations are based totally on the metaconstructs. That is, each function makes all decisions -- such as the legitimacy of a deletion -- based on which metaconstruct was used to generate that particular segment. Their emphasis is on error prevention (i.e., do not let the user create incorrect syntax) and performing any necessary restructuring of

* See Chapter 3 for an expanded discussion of this issue.

** The facility described in this report is called the Program Constructor and Modifier and is abbreviated PCM.

PCMREP. For example, in ALGOL, if the programmer wants to delete the ELSE clause of an IF statement, that certainly is legitimate. But if that IF statement were the THEN part of another IF statement, then its ELSE clause must become the ELSE clause of the inner IF statement.[⊛] That is, (using square brackets as metasymbols to show association)

```
[ IF A THEN [ IF B THEN C ELSE D ] ELSE E ];
```

becomes

```
[ IF A THEN [ IF B THEN C ELSE E ] ];
```

This restructuring is accomplished by the PCM. The algorithm for detecting and then rectifying this problem is described totally in terms of the formalism and its associated metaconstructs. All the modification functions are described only in terms of the formalism.

3. Functions to present information and programs to the programmer. The information functions present attributes the PCM knows about the programmer's symbols. This information is given to the user either at his behest -- e.g., by asking for a list of undefined symbols -- or by the PCM when it discovers an inconsistency -- e.g., a warning that a symbol's value possibly is being used before the symbol was assigned a value. There are two functions for presenting program segments to the programmer. The first is a complete printout of the segment in a structured manner. The structuring of the output is decided on the basis of the metaconstructs. The second type of program presentation presents a short form of the program segment. It is designed to show the structure and not the entire program. For example:

```
BEGIN <<FOO ... COUNT>> ; <<WHILE ... 1.314>> END
```

shows a block with two statements. Enclosed in "<<" and ">>" are the first and last lexical units of those statements.

4. There are actually a few functions which deal with the parsing mechanism. The first function is what is generally considered parsing. It produces a partial PCMREP with the special symbol nodes. It is started by the modification functions which give it the goal and the input string. It should be noted here that the modification functions sometimes have to generate their own goals which are not part of the programming language. Consider the alternating sequence.

parm , *parm* , *parm* , *parm*

[⊛] The PCM philosophy concerning modifications to PCMREP is that the structure should be identical to the PCMREP if the programmer had typed in the entire program directly; thus this change in the association of the ELSE clause.

If the programmer requests to insert before a *parm*, then a goal must be generated for a repeating sequence of an ordered sequence of *parm* , . Similarly, if he wants to insert before a , , then it should be , *parm*.● When the parse returns the partial PCMREP, the modification functions adds it appropriately to the total PCMREP. Then the semantic checker is initiated; it adds the access nodes if necessary and stores information concerning attributes of symbols. In this phase are detected parameter passing inconsistencies, undefined symbols, and possible use of symbols before they have values. This mechanism, driven by the modification functions, alleviates much "reparsing." Since structure can be removed, there is also a function which removes the information added to PCMREP by the semantic checker. All three of these functions have correctors in case of an error or inconsistency. These correctors communicate with the user to either confirm a correction or ask his advice about the error if it cannot rectify the situation itself.

This facility, the Program Constructor and Modifier described above, was designed to allow the programmer the freedom of free-form input but maintain the facility to continue trying to prevent errors. If this is not achieved, correctors take over to try to correct the problem and negotiate with the user about its resolution.

The PCM system is a two-stage process. For each programming language a language *expert* is required to describe that language in the PCM formalism and provide a set of command macros. Once this is accomplished, the PCM is available to programmers who program in that language. Figure 13 shows a block diagram of the PCM system.●● The dashed lines represent data access with double-headed arrows showing read-write access.

6.3 CONCLUSIONS

The initial goal of this research has been achieved -- the design of a sophisticated but language-independent facility for program construction and modification which adapts itself on the basis of a language description. That is, the syntactic and semantic concepts needed for this process were generalized and then adaptively applied via the language description.

It does prevent syntax errors and aids in preventing certain semantic errors. It has good presentation functions for both program segments and relational information concerning symbols. It operates "intelligently" and helps the user in error correction when necessary. All of these combine to make the PCM a sophisticated tool for program construction and modification.

Although initially inspired to develop a facility for an advanced programming environment, the

● Using regular expression notation, an alternating sequence is *parm*(, *parm*)* and the repeating ordered sequence is (, *parm*)* or (*parm* ,)*. This is what the insert function must generate.

●● The repeated use of the block USER refers to the same programmer and is only repeated in the diagram for convenience.

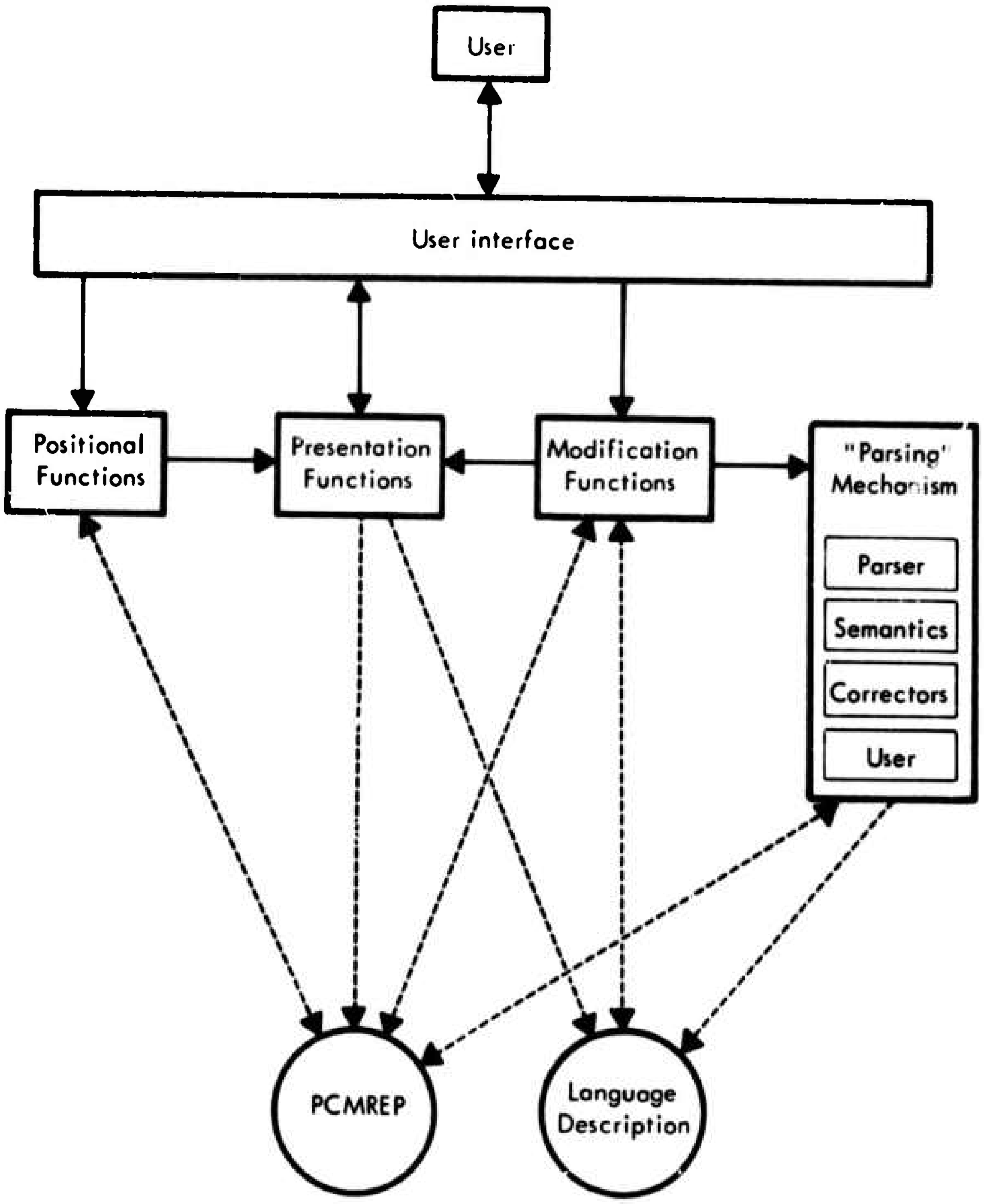

Figure 13. System organization

PCM is a useful facility in its own right. When a programmer constructs a program via PCM, he has a syntactically correct program and some simple semantics have been already checked (e.g., undeclared identifiers). This will prevent extraneous compilations to find these errors, freeing the programmer to find his logical errors.*

To discuss the range of languages applicable to PCM, it must first be divided into syntactic and semantic portions. Syntactically, the PCM formalism is sufficient to describe all reasonable programming languages with two exceptions. PCM does not consider the problem of extensible languages which allow a user to define new syntactic constructs; although it would conceivably be possible to add a language-dependent facility to add new pieces of PCM formalism as the new description was parsed -- similar to the one the compiler for that language uses. The second exception is context-dependent constructs such as PL/1's labelled blocks which allow an END to "closeoff" several nested BEGINs. Semantically, PCM is oriented towards high-level programming languages with scoped and declared variables. The semantic portion of PCM would be basically inoperative on programming languages without this feature -- although the syntactic portion would still operate. But all of the above fall into the category of lack of redundancy of syntactic constructs which directly reflect semantic ones. In a paper discussing the impact of language design on the production of reliable software, this lack of redundancy was considered harmful and was argued against -- backed up by empirical data -- for future language designs (see [Gannon, et al. 75]).

6.4 *FUTURE RESEARCH*

Experiments

The cost of using such a system for program construction is unclear at this time. The prototype system was not implemented with efficiency as a consideration and thus executes very slowly.** It is believed by the author that even using an efficient implementation of the PCM it will take more computer time to construct a program than with a normal text editor. But he also believes that the benefits gained from having a syntactically correct program will override the extra expense during this phase of program development. The author suggests, given an efficient implementation of PCM, that a programmer performance and cost study comparing the two methods of program development up to the point of a successful compilation would be extremely interesting. Trends in the economics of machine time versus human time should probably be considered in the analysis of this study.

Technical Additions

Currently, the PCM only used data types for information presentation to the user. Type checking could be added to the PCM by providing conversion tables, knowledge of the results of operations, and the order of evaluation.

* It should also be extremely useful for programmers who are learning a new programming language.

** Also, the system was implemented in LISP and ran completely interpretively.

SUMMARY AND CONCLUSIONS

PCMREP is a sufficiently richly structured data base that it could be used by other sophisticated facilities. In fact, it has been suggested by one member of USC's Information Sciences Institute Program Verification Project that PCMREP is a good possible replacement to their current program-holding data base.

With the implementation of a language-dependent interpreter and debugger which operated on PCMREP, the birth of a programming environment could be achieved. And since the program's syntax is directly reflected in PCMREP and the formal description of the language is available, the debugger could communicate with the programmer in the syntax of the particular programming language.

David Wilczynski, in his recent dissertation on automatic debugging based on data access [Wilczynski 75], has several heuristics for rewriting a program in an attempt to correct the error. He claims this is possible because of his high-level knowledge of the syntax of his programming language and the relationship between this syntax and the associated control flow. It would be interesting to see if his approach could be applied to some normal high-level programming languages and then generalized using the PCM formalism and execution information to perform LIBNI automatic debugging.

Just as the advent of high-level programming languages was a fundamental step forward in computer science, it is now time for another step forward. A complete set of facilities must be designed which really *aid* the programmer in his tasks if we are ever going to proceed to the next plateau of programming complexity. A true programming environment should be an ever-expanding system which changes as the expertise in any area discussed above increases. With such a system the complexity of the task domains will also increase, and that is the goal of computer science.

REFERENCES

Atwood, J.W.; Holt, R.C.; Horning, J.J.; and Tsichritzis, D. *Proposal for a Project Named SUE.* Computer Systems Research Group, University of Toronto, 1971.

Balzer, R.M. *Language-Independent Programmer's Interface.* University of Southern California, Information Sciences Institute. ISI/RR-73-15. 1974.

Bobrow, D.G. *Requirements for Advanced Programming Systems for List Processing.* Bolt Beranek & Newman Report No. 2339, 1972.

Bratman, H.; Martin, H.G.; and Perstein, E.C. "Program Composition and Editing With an On-line Display." *AFIPS Conference Proceedings.* Vol. XXXIII, Part 2. Montvale, New Jersey: AFIPS Press, 1968, pp. 1349-1360.

Deutsch, L.P. *An Interactive Program Verifier.* Xerox Palo Alto Research Center, Report CSL-73-1, 1973.

Engelbart, D.C. *Advanced Intellect-Augmentation Techniques.* Stanford Research Institute, SRI Project 7079, 1970.

Forsythe, A.I.; Keenan, T.A.; Organick, E.I.; and Stenberg, W. *Computer Science: A First Course.* New York: John Wiley & Sons, 1969.

Gannon, J.D., and Horning, J.J. "The Impact of Language Design on the Production of Reliable Software." *Proceedings of the International Conference on Reliable Software.* 1975, pp. 10-22.

Goldstein, I.P. "Pretty-Printing: Converting List to Linear Structure." Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo No. 279, 1973.

__________. *Understanding Simple Picture Programs.* Massachusetts Institute of Technology, Artificial Intelligence Laboratory. AI-TR-294. 1974.

Good, D.I.; London, R.L.; and Bledsoe, W.W. "An Interactive Program Verification System." *Proceedings of the International Conference on Reliable Software.* 1975, pp. 482-492.

Hansen, W.J. *Creation of Hierarchic Text With a Computer Display.* Argonne National Laboratory, Report No. ANL-7818, 1971.

Hewitt, C.; Bishop, P.; Greif, I.; Smith, B.; Matson, T.; and Steiger, R. "Actor Induction and Meta-evaluation." *ACM Conference on Principles of Programming Languages.* Boston, Mass., 1973, pp. 153-168.

Hopcroft, J.E. and Ullman, J.D. *Formal Languages and Their Relation to Automata.* Reading, Mass.: Addison-Wesley Publishing Company, 1969.

Irons, E.T. "An Error-Correcting Parse Algorithm." *Communications of the ACM.* Vol. VI, No. 11 (November, 1963) pp. 669-673.

James, L.R. *A Syntax Directed Error Recovery Method.* Computer Systems Research Group, University of Toronto, Technical Report CSRG-13, 1972.

Jensen, K. and Wirth, N. *Pascal -- User Manual and Report.* Lecture Notes in Computer Science. Vol. XVIII. Edited by G. Goos and J. Hartmanis. Berlin: Springer-Verlag, 1974.

Johns, C.B. *The Generation of Error Recovering Simple Precedence Parsers.* Computer Science Technical Report No. 74/10. McMasters University, 1974.

Kay, A.C. "The Reactive Engine." Unpublished Ph.D. dissertation, University of Utah, 1969.

King, J.C. "A New Approach to Program Testing." *Proceedings of the International Conference on Reliable Software.* 1975, pp. 228-233.

LaFrance, J.E. *Syntax-Directed Error Recovery for Compilers.* Department of Computer Science, University of Illinois, Report No. 459. 1971.

Lang, D.E. *STYLE Editor: User's Guide.* Department of Computer Science, University of Colorado, 1972.

Lasker, D.M. *An Investigation of a New Method of Constructing Software.* Computer Systems Research Group, University of Toronto, Technical Report CSRG-38, 1974.

Leinius, R.P. "Error Detection and Recovery for Syntax Directed Compiler Systems." Unpublished Ph.D. dissertation, University of Wisconsin, 1970.

Levy, J. *Automatic Correction of Syntax Errors in Programming Languages.* Department of Computer Science, Cornell University, TR-71-116, 1971.

Mitchell, J.G. "The Design and Construction of Flexible and Efficient Interactive Programming Systems." Unpublished Ph.D. dissertation, Carnegie-Mellon University, 1970.

Morton, K.W. "What the Software Engineer Can Do for the Computer User." *Advanced Course on Software Engineering.* Edited by F.L. Bauer. Lecture Notes in Economics and Mathematical Systems, Vol. LXXXI. Berlin: Springer-Verlag, 1973.

Peterson, T.G. "Syntax Error Detection, Correction and Recovery in Parsers." Unpublished Ph.D. dissertation, Stevens Institute of Technology, 1972.

Sussman, J.S. *A Computational Model of Skill Acquisition.* Massachusetts Institute of Technology, Artificial Intelligence Laboratory. AI-TR-297. 1973.

Suzuki, N. "Verifying Programs by Algebraic and Logical Reduction." *Proceedings of the International Conference on Reliable Software.* 1975, pp. 473-481.

Swinehart, D.C. *COPILOT: A Multiple Process Approach to Interactive Programming Systems.* Stanford Artificial Intelligence Laboratory. Memo AIM-230, 1974.

Teitelman, W. "Toward a Programming Laboratory." *International Joint Conference on Artificial Intelligence.* Edited by D. Walker. 1969.

____________. *INTERLISP Reference Manual.* Xerox Palo Alto Research Center, 1974.

von Henke, F.W., and Luckham, D.C. "A Methodology for Verify Programs." *Proceedings of the International Conference on Reliable Software.* 1975, pp. 156-164.

Wegbreit, B. "The ECL Programming Systems." *AFIPS Conference Proceedings.* Vol. XXXVIIII. Montvale, New Jersey: AFIPS Press, 1971, pp. 253-262.

____________. *Multiple Evaluators in an Extensible Programming System.* Center for Research in Computing Technology, Harvard University, ESD-TR-73-112, 1973.

Wilczynski, D. "A Process Elaboration Formalism For Writing and Analyzing Programs." Unpublished Ph.D. dissertation, University of Southern California, 1975. Published as ISI/RR-75-35, October 1975.

Winograd, T. "Breaking the Complexity Barrier **again**." *SIGPLAN Notices.* Vol. X, No. 1 (January, 1975) pp. 13-22.

Youngs, E.A. "Human Errors in Programming." *International Journal of Man-Machine Studies.* Vol. VI, No. 6 (May, 1974) pp. 361-376.